University of Zurich[UZH]

Department of Informatics

Getting the Right Code Changes Done Right

Dissertation submitted to the Faculty of Business,
Economics and Informatics
of the University of Zurich

to obtain the degree of
Doktorin der Wissenschaften, Dr. sc.
(corresponds to Doctor of Science, PhD)

presented by
Katja Kevic
from Nesslau, SG, Switzerland

approved in September 2017

at the request of
Prof. Dr. Thomas Fritz, University of Zurich, Switzerland
Prof. Dr. Harald C. Gall, University of Zurich, Switzerland
Prof. Lori L. Pollock, Ph.D., University of Delaware, USA

University of Zurich[UZH]

The Faculty of Business, Economics and Informatics of the University of Zurich hereby authorizes the printing of this dissertation, without indicating an opinion of the views expressed in the work.

Zurich, September 20, 2017

Chairwoman of the Doctoral Board: Prof. Dr. Elaine M. Huang

Acknowledgements

First and foremost, I thank my advisor Thomas Fritz. He continuously supported me with his excellence and enthusiasm during the time of my PhD and always encouraged me to pursue my ideas. I also thank Harald Gall for his precious advice and support and for being part of my PhD committee. Even though he has a fully loaded calendar, he always found time for a discussion. I thank Lori Pollock for spending her valuable time to examine my thesis and serving on my PhD committee.

During the time of my PhD I had the great pleasure to work with Brendan Murphy and David Shepherd. I thank them for the interesting projects, discussions and their great support.

Special thanks go to the current and past members of the Software Evolution and Architecture Lab, who made the time of my PhD memorable: Carol Alexandru, Adelina Ciurumelea, Giovanni Grano, Christoph Laaber, Philipp Leitner, André Meyer, Sebastiano Panichella, Sebastian Proksch, Carmine Vassallo, Chaman Wijesiriwardana, Manuela Züger, Pooyan Behnamghader, Martin Brandtner, Emanuel Giger, Christian Inzinger, Michael Würsch, and in particular Jürgen Cito, Sebastian Müller, and Gerald Schermann. Working with you was great. I appreciate the fun times we had in the office and on conference trips and the valuable feedback they provided on my work.

I thank my family, Venla, Albert, Tanja, Maja, Juri, and Christian and my dear friends Patrizia, Luana, and Désirée for their support and encouragement during the time of my PhD. Last but not least, I would like to thank Amedeo, who supported me throughout the time of my PhD and especially before paper deadlines.

Katja Kevic
Zürich, September 2017

Abstract

Successful software undergoes constant change for reasons such as evolving preferences of the users, changing laws, or advanced hardware. For many software projects the time-to-market of software changes is a crucial competitive advantage, requiring software changes to be realized rapidly. At the same time, successful software projects grow in terms of source code and people, making software changes more difficult. To be successful in this fast-paced and challenging setting, developers need to focus on implementing software changes efficiently as well as on spending their time effectively, i.e. on the changes that are important to the user.

In this thesis we address this challenge of delivering the right (user-wanted) changes right (efficiently). In particular, we focus on two aspects: (a) the efficient execution of software changes by software developers in the source code, (b) the effective development of software changes by using experimentation to determine the user-wanted changes.

Over the past years, various software engineering tools emerged that help developers perform changes in the source code more efficiently, ranging from systems that recommend relevant artifacts for a given change task, to tools that summarize source code elements to speed up code comprehension. All of these tools are based on an understanding and a model of developers' navigation behavior. The more we know about developers performing change tasks, the better we are able to understand the difficulties developers face and the better we are able to provide tool support for developers during their work in the source code. Yet, relatively few studies have been undertaken that investigate the fine-grained

navigation behavior of developers in the source code. In a series of exploratory studies and empirical analyses we gathered observations on developers' navigation behavior and devised a model that captures the fine-grained characteristics of developers' navigations. Our detailed analysis of developers' navigation behavior in the source code replicates and extends insights from other researchers about the navigation behavior of developers. Our results provide evidence that the accuracy and granularity of existing software engineering tools can be improved through our observations by enabling the recognition of carried-out developer-activities and relevant code elements for change tasks with high accuracy.

To be successful and remain competitive, it is also important that the development efforts of the team are spent effectively, i.e. on the software changes that matter to the users. Today's capability to rapidly release software to users, bares the possibility to rapidly learn about the preferences of the users through experimentation with code changes. The more we know about such an experimental procedure, the better we are able to increase the effectiveness of development teams during the development of their product. Yet, while many companies perform some form of experimentation, very few studies empirically characterized the full life-cycle of an experiment, from the code changes all the way to the analysis of the captured user preferences. To better understand challenges that arise within an experimental procedure and the characteristics of the code changes that are associated with experiments, we conducted a large-scale data analysis on the experimental procedure of a mature product and its associated code changes. The results of the analysis reveal insights into the time spans associated to experiments, as well as into the efficiency of such an experimental procedure. The results of the analysis further illustrated differences between the code changes that are appreciated by the users and the code changes that have not been appreciated by the users.

Overall, the results from our studies and analyses open new research directions to support developers in delivering efficiently realized and user-wanted software changes. For example through explicitly integrating information about experiments in the integrated development environment, providing developers direct feedback about the impact of their code changes.

Zusammenfassung

Erfolgreiche Software unterliegt ständiger Veränderung, die unter anderem hervorgerufen wird durch veränderte Benutzerpräferenzen, eine sich ändernde Gesetzeslage oder verbesserte Hardwarekomponenten. Für viele Softwareprodukte ist die Einführungszeit von Softwareänderungen ein entscheidender Wettbewerbsvorteil, was dazu führt, dass Softwareänderungen schnell realisiert werden müssen. Zugleich werden Softwareänderungen jedoch immer schwieriger umsetzbar, da der Quellcode und die Anzahl der Softwareentwickler in erfolgreichen Projekten wächst. Um in diesem schnelllebigen und herausfordernden Umfeld erfolgreich zu sein, müssen sich Entwickler darauf konzentrieren, die Softwareänderungen effizient zu realisieren und ihre Zeit effektiv zu investieren, d.h. in jene Softwareänderungen, die wichtig sind für die Benutzer des Softwaresystems.

In dieser Arbeit befassen wir uns mit der Herausforderung, die richtigen (anwenderorientierten) Softwareänderungen richtig (effizient) an Benutzer zu liefern. Insbesondere fokussieren wir uns auf zwei Aspekte: (a) die effiziente Durchführung der Softwareänderungen von Entwicklern im Quellcode, (b) die effektive Durchführung von Softwareänderungen, wobei Experimente durchgeführt werden, um jene Softwareänderungen zu ermitteln, die von den Benutzern geschätzt werden.

In den vergangenen Jahren wurden verschiedene Werkzeuge entwickelt, die Softwareentwicklern helfen, Änderungen im Quellcode effizienter zu gestalten. Solche Werkzeuge reichen von Systemen, die relevante Artefakte für eine gegebene Quellcodeänderungsaufgabe empfehlen, bis hin zu Werkzeugen, die Quellcodeelemente zusammenfassen, um das Verstehen des Quellcodes zu beschleunigen. Alle

diese Werkzeuge beruhen auf dem Verständnis und einem Modell des Navigationsverhaltens der Entwickler im Quellcode. Je mehr wir über die Durchführung von Quellcodeänderungsaufgaben von Entwicklern wissen, desto besser können wir deren Schwierigkeiten verstehen und Werkzeuge realisieren, die die Entwickler bei ihrer Arbeit im Quellcode unterstützen. Es wurden jedoch relativ wenige Studien durchgeführt, die das feingranulare Navigationsverhalten von Entwicklern im Quellcode untersuchen. In einer Reihe von explorativen Studien und empirischen Analysen haben wir Erkenntnisse über das Navigationsverhalten von Entwicklern im Quellcode gesammelt und ein Modell formuliert, das das feingranulare Navigationsverhalten von Entwicklern erfasst. Unsere detaillierte Analyse des Navigationsverhaltens der Entwickler im Quellcode repliziert und erweitert Erkenntnisse von anderen Forschern über das Navigationsverhalten von Entwicklern. Unsere Ergebnisse belegen, dass die Genauigkeit und Granularität bestehender Entwicklerwerkzeuge durch unsere Beobachtungen verbessert werden kann, indem die Erkennung von Entwickleraktivitäten und relevanten Quellcodeelementen für Quellcodeänderungsaufgaben mit hoher Genauigkeit ermöglicht wird.

Um erfolgreich und kompetitiv zu bleiben, ist es wichtig, dass der Arbeitsaufwand der Entwickler effektiv investiert wird, d.h. in jene Softwareänderungen, die von den Benutzern geschätzt werden. Die heutige Fähigkeit, Software schnell an Benutzer freizugeben, birgt die Möglichkeit, Experimente mit den Softwareänderungen durchzuführen und so schnell etwas über die Benutzerpräferenzen zu lernen. Je mehr wir über ein solches experimentelles Verfahren wissen, desto besser können wir die Effektivität der Entwicklerteams bei der Entwicklung ihres Produktes erhöhen. Obwohl viele Firmen verschiedene Formen von Experimenten mit ihren Softwareänderungen durchführen, gibt es sehr wenige empirische Studien, die den vollen Lebenszyklus eines Experimentes nachvollziehen, von den Softwareänderungen bis hin zur Analyse der erfassten Benutzerpräferenzen. Um die Schwierigkeiten in einem solchen experimentellen Prozess und die Charakteristiken der Softwareänderungen solcher Experimentes besser zu verstehen, haben wir eine grosse Datenanalyse eines experimentellen Prozesses eines ausgereiften Produktes und dessen Softwareänderungen vorgenommen. Die Resultate der

Analyse gewähren Einblicke in die Dauer der Durchführung solcher Experimente sowie über die Effizienz eines solchen experimentellen Verfahrens. Die Resultate der Datenanalyse zeigen ferner Unterschiede zwischen den Softwareänderungen, die von den Benutzern geschätzt werden, und den Softwareänderungen, die von den Benutzern nicht geschätzt werden.

Die Ergebnisse unserer Studien und Datenanalysen eröffnen insgesamt neue Forschungsrichtungen, um Entwickler bei der Bereitstellung effizient realisierter und anwenderorientierter Softwareänderungen zu unterstützen. Zum Beispiel durch die explizite Integration von Informationen über Experimente in die integrierte Entwicklungsumgebung, was den Entwicklern ein direktes Feedback über die Auswirkungen ihrer Softwareänderungen bieten würde.

Contents

List of Figures

List of Tables

1

Synopsis

To be successful, software development companies have to continuously make changes to their software, for instance, to adapt to evolving user preferences or new hardware [Brooks, 1987]. Especially in a context where time-to-market is crucial to remain competitive [Parnin et al., 2017], companies have to make and deliver these changes in a timely manner. Given the growing complexity of software [Storey et al., 1997] and the scarcity of developers' time [Nan and Harter, 2009], staying successful is challenging at best. Therefore, it is important that (a) developers' time on code changes is spent as efficient as possible and that (b) their time is spent on the code changes the user wants.

To make changes to a software system, developers need to identify the parts in the source code relevant for the change. As the relevant code for a change is often scattered across the system [Kiczales et al., 1997], developers spend a substantial amount of their time searching and navigating to identify the relevant code for the change [Ko et al., 2006]. To support developers becoming more efficient during this source code exploration phase, researchers have studied developers'

behavior and created approaches based on these studies that help to identify the relevant code for the change. For instance, by studying developers' navigation behavior, tools emerged that point developers directly to relevant source code parts [Robillard, 2005] or other relevant artifacts [Ponzanelli et al., 2014, Čubranić and Murphy, 2003], provide a task-focused user interface [Kersten and Murphy, 2006], or provide code summaries [Moreno et al., 2013]. However, little is known about the more fine-grained source code navigation, such as the changing cues that developers follow in the source code over the course of a change task, or the source code that developers look and focus on, or characteristics of developers' navigation behavior for different activities that developers carry out to perform a change task. A better understanding of developers' navigations could help with improving the identification, recommendation and summarization of task-relevant source code elements with an increased accuracy and granularity. We therefore examine the fine-grained navigation behavior of developers when performing realistic change tasks. We investigate different activities that developers engage in and further use eye tracking to capture more and more-fine grained information about the developers' navigation behavior in the source code.

To be successful and focus developers' time on the relevant changes, development teams need to ensure that the changes to the software are the ones that the customer wants. Previously it was difficult for development teams to assess the value of a code change a priori [Kohavi et al., 2009a]. Through the adoption of development techniques that allow to continuously release changes and gather users' interactions with the changes, development teams have nowadays the capability to learn about their users' behavior and therefore better estimate the impact of a given change. Only few researchers investigated this experimental procedure that is applied to code changes. These researchers primarily focus on experience reports, lessons learned [Kohavi et al., 2009a, Lindgren and Münch, 2015] or investigate specific statistical means to improve the execution of such code experiments [Deng et al., 2013]. To the best of our knowledge, no other work in the field empirically investigated the full life-cycle of an experiment from the code change to the analysis of the gathered users' interactions with the change in the product. A better understanding of the characteristics of the

code changes that are used in experiments and the time that is needed to run an experiment, might enable development teams to better estimate the user impact of code changes a priori and adapt their product strategies accordingly. We fill this gap by empirically analyzing the time spans and development efforts of code experiments that have been conducted in a large-scale and mature software product.

In summary, in this work we are looking at effective and efficient development of code changes. In particular, we are investigating developers' navigation behavior when they perform change tasks and the experimentation process with code changes. A better understanding of these two aspects can inform better support tools to increase the efficiency and effectiveness of developers in their work on code changes. Our two hypotheses are:

> ***H1: A model based on fine-grained characteristics of developers' navigation behavior allows to automatically detect carried-out activities and relevant code elements for change tasks with high accuracy.***

> ***H2: A model based on the characteristics of experiments and code changes allows to identify bottlenecks and features of successfully shipped code changes in the experimentation process.***

These models can directly help to support developers in their work on change tasks, for instance through developer support tools for efficient source code navigation and approaches to optimize the delivery of effective code changes. We investigate hypothesis H1 in a series of *exploratory studies* and *data analyses* in which we examine developers' fine-grained navigation behavior for real change tasks in real software systems. To investigate hypothesis H2 we explore an experimentation process that is used in a big and mature software system and conduct a *large-scale data analysis* of its experimentation process.

In our research, we focus on three research questions that are described in Chapter 1.1. Chapter 1.2 provides an overview of the approach that we used to

answer our research questions and Chapter 1.3 provides a summary of our findings. We further elaborate on the threats to validity of our approach in Chapter 1.4, potential future work in Chapter 1.5, and related work in Chapter 1.6. We summarize the contributions of our work in Chapter 1.7, and provide a roadmap of this thesis in Chapter 1.8.

1.1 Research Questions

In this research, we investigate how to support developers in working efficiently on the user-wanted code changes. We organize our research along two axes (see Figure 1.1): one axis denotes the development and deployment of code changes and the second axis denotes the granularity of the captured interactions with the source code for a change task.

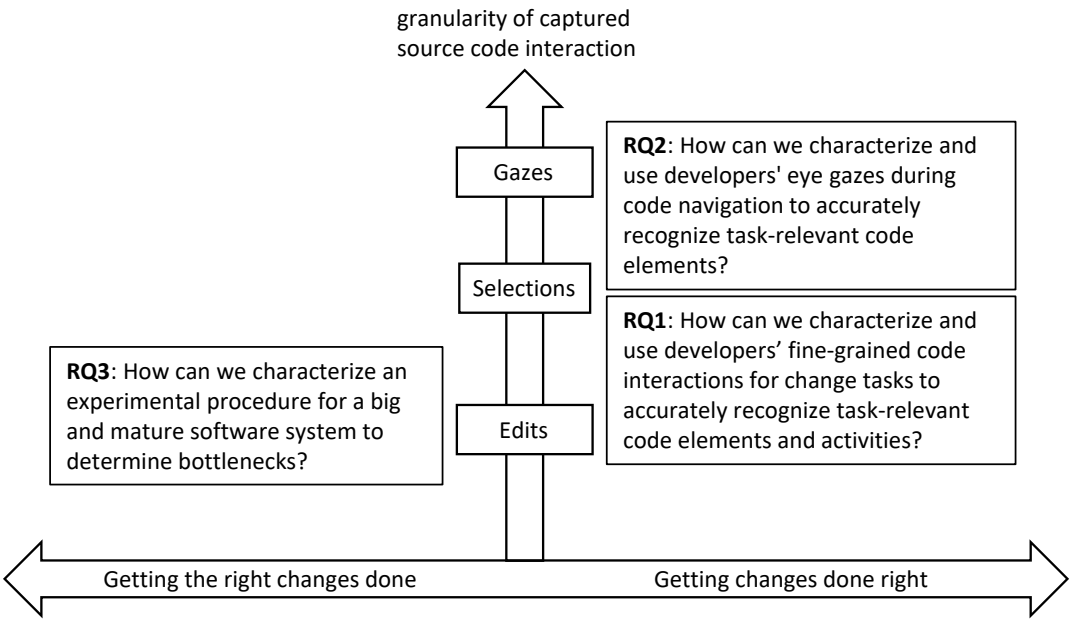


Figure 1.1: Within this thesis, we focus on the efficient execution of software changes and on the execution of the effective software changes.

One challenge that developers face is that task-relevant source code is often distributed across various code elements, such as classes, methods, and statements [Kiczales et al., 1997] and maintaining a mental model of these source code parts is challenging [LaToza et al., 2006]. Researchers have studied developers' navigation behavior by observing them working on small change tasks, e.g. [Ko et al., 2006], or even used eye-tracking to capture their eye gazes, e.g. [Rodeghero et al., 2014]. Yet little is known about developers' fine-grained code navigation on real change tasks. In our research, we address this gap by looking at developers' fine-grained navigation behavior, including their eye gazes, while working on real change tasks of a reasonable sized software system. For a more fine-grained model of developers' code navigation and better approaches to help developers perform change tasks efficiently, we investigate the following two research questions:

RQ1: How can we characterize and use developers' fine-grained code interactions for change tasks to accurately recognize task-relevant code elements and activities?

RQ2: How can we characterize and use developers' eye gazes during code navigation to accurately recognize task-relevant code elements?

A second challenge in the development and delivery of software changes is to ensure that the user-wanted changes are being made. With the increasing rate of releasing software, development teams have the capability to rapidly test whether users react positively to a code change. Through instrumenting the product and exposing randomly chosen user groups to different versions of the product, development teams can run A/B-Tests with every code change. Based on the captured users' reactions, a development team can then make a data-driven decision on whether they want to focus their development efforts on the change to improve it and deploy it to all users or abandon the code change.

Other researchers investigated in this context the challenges that organizations face when they adopt to systematically experiment with code changes [Davenport, 2009, Kohavi et al., 2009b, Kohavi et al., 2013, Lindgren and Münch, 2015]. They

identified cultural shifts, slow development cycles, product instrumentation, and the identification of metrics to quantify user experience as challenges for organizations which conduct experiments with their code changes. Further work in this field, describes a model for an experimentation process that is composed into build-measure-learn blocks [Fagerholm et al., 2014, Fagerholm et al., 2017]. The model of Fagerholm et al. shows the interdependency between experiments and product strategies.

Compared to previous works, we are interested in exploring the full life-cycle, from the first code commit to the analysis of the captured data. Knowing more about the characteristics of an experiment, might enable product teams to plan their product strategies more efficiently. Furthermore, we are interested in understanding more about those code changes that are involved in an experiment. Knowing more about the code changes of successful experiments might enable us to recognize them early on in the development cycle. Our third research question is:

> ***RQ3: How can we characterize an experimental procedure for a big and mature software system to determine bottlenecks?***

These insights can help development teams to better plan product strategies and might enable to detect user-wanted code changes early on.

1.2 Research Approach

To explore and answer our research questions (see Chapter 1.1), we conducted four exploratory studies and three data analyses (see Table 1.1). Study 2a is a follow-up study of study 2 that was conducted with a subset of the participants of study 2. In our exploratory studies, we observed a total of 26 professional developers and 29 student or faculty developers working on open- and closed-source software systems.

We followed a similar approach in all of our exploratory studies. First, we recruited study participants either through e-mail or personal contacts. We asked

Table 1.1: Overview of the exploratory studies and data analyses that we conducted to answer the research questions. (*Prof.* denotes professional developers, *Stud.* denotes student developers, and *Fac.* denotes faculty developers.)

Exploratory Studies

RQ	*ID*	*Topic*	*Lab*	*Field*	*#Prof.*	*#Stud./Fac.*	*Chapter*
1	1	Nav. Steps	✓		5	7	2
1	2	Activities	✓	✓	9	12	3
2	3	Gazes	✓		12	10	4
2	2a	Gaze Relevancy	✓			7	5

Data Analyses

RQ	*ID*	*Topic*	*Artifact*	*Chapter*
1	1a	Nav. Steps	2253 change tasks	2
1	1b	Nav. Steps	8000 hours of development activity	2
3	2	Experiments	21,220 experiments and code changes	6

the recruited developers to either work on change tasks of an open-source system that we gave them or on their own change tasks at their work place. In all studies, we captured developers navigation steps within the source code. Thereby, we incrementally increased the level of granularity that we captured about the developers' navigation steps in successive studies. The captured data allowed us to make observations about developers' fine-grained navigation behavior. Based on these observations we inferred design requirements for developer support tools and devised approaches to accurately recognize task-relevant code elements, as well as activities that are carried out by developers.

Subject Systems. We asked the participants of our studies to either work on their own work projects (study 2) or we selected an open-source system to work on (studies 1-3). We selected the open-source systems used in our studies based on their general availability and their active use and maintenance. The open-source systems used in our studies cover a broad range of application domains, from

Json file (de-)serializers, to bibliographic database management systems, to time trackers. The selected open-source systems further differ in their sizes, ranging from 13.5k non commented lines of code (NCLOC) spread across 167 top level classes to systems containing 52.5k NCLOC spread across 439 classes. The study participants' work systems in the field study had on average 244k of code. All subject systems used in our exploratory studies were written in Java.

Data Collection. In all our studies we captured developers' navigation within their IDE. Thereby, we increased the granularity of the data captured. In our first exploratory study, we captured developers' screens and manually transcribed their navigation behavior between classes and methods. In the subsequent exploratory studies, we automatically captured developers' clicks within their IDE either through the use of Mylyn [Mylyn, 2015, Kersten and Murphy, 2005] or self-built monitoring systems. The click monitoring systems that we used captured developers' navigations on class and method level as well. For study 2 we further captured self-reported descriptions about the activities the developers worked on, as well as relevancy assessments for code elements they explored. For study 2a and 3 we further captured developers' eye gazes within their IDE. Using a software called iTrace [Shaffer et al., 2015] enabled us to link the captured eye gazes directly to AST nodes of the software system that the developers looked at. While we used a high-end eye tracker for study 3 (Tobii X60), we used a comparatively low-cost eye tracker (Eye Tribe ET1000) in study 2a.

In our data analyses we analyzed both, closed- and open-source software artifacts from hundreds of developers. The primary focus of the data analyses 1a and 1b was to validate the observations from the exploratory study 1. In total, we analyzed in data analyses 1a and 1b the navigation steps within software systems of 80 developers, whereas data analysis 1a is based on the developers' work in an open-source system and data analysis 1b is based on developers' work in a closed-source system. To investigate our third research question, we conducted a third data analysis. Data analysis 2 is a large-scale data analysis in which we analyzed three different artifacts that are related to the experimental procedure that is applied in a big, mature, and closed-source software system. In particular, we analyzed 21,220 experiments that have been conducted within

Table 1.2: Observations from exploratory study 1 and inferred design implications.

#	Empirical Observation	Design Requirement
O1	Code navigation exhibits a high structural & lexical cohesion	Combine structural and lexical navigation support and provide proactive structural and lexical context
O2	Lexical cohesion of code navigation models is stronger at beginning of the exploration	Adapt navigation support over time
O3	Developers use a combination of search and structural navigation	Combine search and navigation
O4	Code navigation is hardly influenced by the size of the actual change, but differs substantially by developer	Provide personalized navigation support
O5	Change task type influences the number of code elements explored	Tailor support to change task type

the past 2.5 years, the collected users' reactions for each experiment, and the code changes associated to each experiment.

1.3 Findings

In the following, we summarize the findings of our exploratory studies and data analyses with respect to our research questions (see Chapter 1.1). More details about these findings can be found in Chapters 2-6.

1.3.1 Developers' Navigation Behavior (RQ1)

Our exploratory studies 1 and 2 (see Table 1.1) resulted in a series of empirical observations about developers' navigation behavior for change tasks (see Table 1.2), as well as for particular activities that developers carry out to perform a change task.

Navigation Behavior for Change Tasks. Analyzing developers' navigation behavior over the course of a whole change task provides further evidence on previous research that developers navigate extensively between code elements that have structural and lexical relations (O1 in Table 1.2). We extend the existing research in that we observed that developers navigated around groups of code elements that are structurally connected through call dependencies, usage or implements relations. We further observed that the code elements not belonging to this large group of connected code elements were explored towards the beginning of the change task exploration. We found that developers who did more structural navigation finished the change task sooner (Pearson's $r = -.49$). We thereby extend previous research [Robillard et al., 2004] that found that successful developers do more structural navigation. We also confirm existing research [Lawrance et al., 2008, Ko et al., 2006] which found that developers follow lexical cues when exploring source code. We extend this existing research in that we observed that developers followed lexical cues in particular in the beginning of the change task exploration and that lexical cues are particularly important when switching between different classes (O2).

Over all developers we found that developers use a combination of search and structural navigation (O3) and that the navigation for a change task differs substantially across developers (O4) and for different change task types, such as bugs, enhancements or minor issues (O5).

Based on these empirical observations, we suggest design implications for developer support tools, as shown in Table 1.2. To examine the value of our insights and to address the design requirements inferred from O1 and O3, we developed an approach called CoMoGen [Kevic et al., 2014]. CoMoGen is a code search engine that takes into account structural and lexical relations of the source code and combines search and navigation in that search results are depicted with contextual information. In a preliminary user study that we conducted, we found that CoMoGen supports developers in better understanding and assessing the relevance of search results and that less navigation is required to perform a change task.

Developers' Activities. We found a set of reoccurring activity types into which multiple developers decomposed their change tasks into. We further found that activities depict an atomic unit of work that is independent from individual developers and task types. A developer's activity is relatively small, comprising 8.7 unique code elements on average and only about a third of the code elements explored for an activity are perceived relevant by the developers. Our results further show that developers navigate differently for different activities. These differences in their navigation behavior allow on the one hand to identify the type of the activity that they worked on and on the other hand, allow to detect switches between activities with high accuracy. Through these fine-grained characteristics of developers' navigations in the source code that allow to automatically recognize activities, navigation support tools tailored to particular activities can be provided. We further showed in a small experiment that the information about developers' activities can improve the automatic recognition of task-relevant code elements considerably (precision is improved by 32.59% and recall is improved by 57.14%).

Overall, our findings provide evidence for our hypothesis H1 that a fine-grained analysis of developers' navigation behavior can be used to automatically detect carried-out activities and task-relevant code elements. More details about our results can be found in Chapters 2 and 3.

1.3.2 Developers' Eye Gazes (RQ2)

The results from our third exploratory study (study 3), in which we captured developers' clicks and eye gazes, show that developers look at substantially more source code elements than they click on. In particular, we found that through the use of eye trackers significantly more methods and more method switches are captured. Gaze-based navigations and click-based navigations further capture different aspects about developers' navigation behavior in the source code as the methods on which developers look at the longest do not match with the methods that were selected most frequently by the developers.

Between Method Navigation. We observed that gaze-based navigation and click-based navigation capture a different image of the developers' navigation behavior with respect to structural and proximal relations between successively visited methods. In particular, we found that methods developers successively looked at are less often connected through call dependencies ($M = 4.1\%$) than methods developers clicked on ($M = 22.61\%$). We further observed that the proximity of methods is an important aspect for the navigation between methods. We found for the gaze-based method switches that developers switched in 36.95% of the cases to methods located right above or underneath and for the click-based method switches we found that developers clicked in 69.93% of the cases on a method right above or underneath.

Within Method Navigation. We found that developers only look at few lines within methods ($M = 32.16\%, SD = 24.95\%$), that they switch often between these lines ($M = 39.95, SD = 100.99$ line switches for each method), and that these lines are connected through same variables. Developers spend most of their time looking at method invocations ($M = 4.1s$) and variable declarations ($M = 1.8s$). We further found that developers look surprisingly few times at method signatures. In fact, in 46% of all method explorations, the signature was ignored at all. This finding demonstrates that developers read source code differently when they are performing real change tasks in real software systems and are not limited to code snippets in which case the method signature might be more relevant.

Navigation Strategies. Fine-grained eye gaze data enables to study small differences in the way developers explore source code. We identified two different strategies (i.e. the skimming strategy and the seeking strategy) about the way developers explored source code. Developers who skim the source code switch more to methods in close proximity, read overall less lines within methods, and focus on less methods. On the other hand, developers who seek the source code switch to less methods in close proximity, read overall more lines within methods, and focus on more methods. We also found evidence that the information given in the change task description influences which of these two strategies is applied by developers.

Developer Support. Based on our observations about developers' fine-grained navigation behavior, we developed a series of approaches that demonstrate the potential of our observations to improve tool support for developers:

- *Fine-granular line-level navigation recommendations.*
- *Task difficulty prediction.*
- *Automatically recognizing the relevant lines within methods.*
- *Improved recognition of relevant methods.*

Our findings increased the understanding of developers' fine-grained source code navigation for change tasks. We further demonstrated that these fine-grained observations can be used to automatically detect relevant code elements for change tasks. More details about our findings can be found in Chapters 4 and 5.

1.3.3 Developers' Experiments (RQ3)

Our analysis on past experiments that have been conducted within a big and mature software system revealed that the execution of trustworthy code experiments takes a substantial amount of time. In particular, we found that experiments take on average 42 days, from the deployment of the code change until the analysis of the captured metrics about the users' interactions with the code change. Further, for each experiment, there are on average 4.8 people involved which face the challenge of interpreting the 1409 metrics that are on average collected.

Our analysis revealed that, at the time of our analysis, around a third of the experiments have been successful[1]. We also found that this success rate differs substantially among different components of the system. We further found differences in the characteristics of the code changes that were associated to successful experiments and the code changes that are associated to experiments that have not been deployed to all users at the time of our analysis. We found that the successful experiments include significantly more changes and significantly more developers contributing or altering significantly more code.

[1] The success of an experiment can be considered from different standpoints, see Chapter 6.9.4

These results are a first step towards recognizing early in the development cycle the code changes that are more likely to be important for the users of the product. More details about our findings can be found in Chapter 6.

1.4 Threats to Validity

To better understand developers' navigation behavior during change tasks, we conducted three exploratory studies and two data analyses. To better understand how code experiments are conducted within a big and mature system, we conducted a large-scale data analysis on artifacts of experiments and the associated code changes (see Table 1.1). In the following, we discuss the threats to validity of these exploratory studies and data analyses.

External Validity. The generalizability of our exploratory studies is threatened by the limited size of our subject sample and change tasks that were used in our studies. We tried to mitigate this risk by including a wide variety of participants in our studies with different programming experiences. We observed professional developers from different companies of different industrial fields and student or faculty developers from different universities. We further observed developers working on a wide variety of different change tasks. All change tasks in our studies stem either from reasonably sized open-source systems that are actively used and maintained or in the case of the field study, we observed developers working on their assigned change tasks at their work places. Further, our studies are based on one particular IDE (Eclipse) and one programming language (Java). Even though for some professional developers who participated in our studies Eclipse is not their preferred IDE and Java is not their preferred programming language, all of the study participants stated that they did not have problems using Eclipse or Java during the study. Nevertheless, developers' navigation behavior might differ for other programming environments with different navigation support tools and different structures in the programming language. Further studies are needed to generalize our results to developers using another set of development tools.

The generalizability of our data analyses is threatened by data artifacts stemming from one particular project (i.e. data analysis 1a and data analysis 2) or by stemming from one specific company (i.e. data analysis 1a, 1b, and 2). We tried to mitigate this risk by combining our data analyses on the navigation behavior of developers with the exploratory study 1. To strengthen the generalizability of our analysis on experiments (study 2), we analyzed a large-scale project, that consists of several different subcomponents with different characteristics.

Internal Validity. Developers who participated in our lab studies, often solved more than one change task of a software system. Hence, we might observe a learning effect over the course of the study sessions, which might influence the developers' navigation behavior. We tried to mitigate this learning effect by applying a counterbalance measure design. The participants' navigation behavior might be further influenced by knowing that they are observed. Biometric sensors, such as eye trackers, might increase this potential uneasiness more than click trackers do. Participants' uneasiness might be reflected in their perceived task difficulty and the way they look at the code. We tried to mitigate this risk by interacting as little as possible with the study participants during the study time. The internal validity of the results from our exploratory studies and data analyses 1a and 1b is strengthened by the fact that we observed similar navigation behavior across participants with different programming experiences and across different systems. The internal validity of our research is further strengthened as we confirmed results from previous studies of other researchers and further found concordant evidence about the characteristics of the developers' navigation across all of our exploratory studies and data analyses.

Construct Validity. The validity of our results on developers' fine-grained navigation behavior is threatened by the accuracy of the collected eye gaze data since eye trackers do not always remain well calibrated. We tried to mitigate this risk by regularly recalibrating the used eye trackers.

Since for some of the analyzed data artifacts explicit links among them are missing, we had to define procedures or approximations to re-establish those links. For instance, the Blaze data that we analyzed in data analysis 1b does not

contain information on the particular tasks developers worked on. Therefore, we partitioned the data stream into one hour sessions rather than task sessions. We tried to validate our approximation by analyzing a second dataset that explicitly captures information on developers' tasks (data analysis 1a). For data analysis 2, an explicit link between experiments and associated source code changes was not existent and we designed a procedure to re-establish these links. We implemented a parser, enabling us to link experiments to source code changes and to classify each change set in our analysis to one of four fuzzy groups that indicate the intensity of the connection to experiments.

1.5 Opportunities and Future Work

The results from our exploratory studies and data analyses open new research directions, such as task resumption support tools that work on various granularity levels, opportunities for tools that facilitate the experimentation with code changes, and explicit integration of experiment-related information into the developers' IDE. More opportunities and future work is discussed in the Chapters 2-6.

Task Resumption with Adjustable Granularity. In our research, we investigated developers' work in the source code on different granularity levels. In particular, we analyzed developers' interactions with single source code lines all the way to the activities that developers carry out when performing a change task. We envision to create a time-line that integrates these different kinds of granularity levels. In particular, we plan to investigate a time line which depicts the automatically detected activities of developers in combination with the relevant code elements. Developers would have the possibility to zoom into individual activities and get a summary of the relevant code elements. Developers can then decide to zoom in further to find the source code lines that they perceived relevant. While such a time line could help to resume tasks, it can also serve as a retrospective view of the work day. For future work and to extend such a time-line, we further plan to investigate developers' work on more coarse-grained

tasks as well that are not necessarily associated with their work in the source code.

Opportunities for Code Experimentation Support. In our data analysis on the experimental procedure of a large-scale and mature software system, we found that for an experiment on average 1409 ($SD = 488$) different metrics are captured. As these metrics are captured multiple times a day and form trade-offs among another, the interpretation of these metrics is not always straight-forward [Kohavi et al., 2012]. We therefore plan, on the one hand, to investigate summarized views that facilitate the interpretation of such captured metrics. On the other hand, we plan to investigate whether we can learn from previous experiments and their captured metrics to automatically recommend the outcome of an experiment. Additionally, we plan to further investigate the characteristics of the development efforts of experiments that have been fully shipped to inform approaches that recognize early the software changes that are appreciated by the users.

Visibility of Experiments in the IDE. In our research we address the challenge of delivering right software changes to users. Currently, developers' work in the source code is often decoupled from the experiment that is run with those code changes. We therefore suggest to integrate information artifacts of the experimental procedure directly into the IDE of the developers. In particular, we envision to explicitly depict the link between different parts of the source to different experiments. Furthermore, we envision to integrate live information on relevant metrics and live information on the amount of user traffic that is currently exposed to this part of the source code. This integration bears potential, in that it segments the source code in terms of experiments. Similar as task-focused views in IDEs [Kersten and Murphy, 2006], developers could select experiment-focused views. Furthermore, parts in the source code that are affected by multiple experiments become immediately visible and can be monitored more closely. Finally, developers could immediately see the impact of the source code changes that they realized.

1.6 Related Work

Work related to our research can broadly be categorized into exploratory studies that explored developers' work in the source code, models that were implemented to determine the relevancy of code elements for change tasks, and works that explored the use of experiments to find out which code changes to keep or alter in a software system.

1.6.1 Exploratory Studies on How Developers Explore Source Code

Researchers have extensively observed and studied the program investigation behavior of software developers during maintenance tasks. One of the best-known studies in this field has been conducted by Ko et al. [Ko and Myers, 2005,Ko et al., 2006]. In their study they investigated navigation patterns of ten developers who worked on simplistic tasks in a very small system. They observed that developers often start with a search and then navigate to related elements, thereby collecting small fragments of task-relevant code. Sillito et al. [Sillito et al., 2005, Sillito et al., 2006] made a similar observation when they conducted a laboratory and an industrial study with 25 developers in total working either on a given or on one of their own change tasks. They observed that developers often first look for an initial focus point and then explore relationships from these points, also revisiting elements. Starke et al. [Starke et al., 2009] investigated this initial search phase in more detail by observing ten developers performing tasks for 30 minutes. Starke et al. investigated in particular how participants form search queries and then skim through the results. Wang et al. [Wang et al., 2011] also focused on the initial exploration phase and observed 38 students performing tasks. They identified search patterns that were used by the study participants, such as execution-based and exploration-based search. Other researchers investigated differences in the navigation behavior of developers. Robillard et al. [Robillard et al., 2004] found by studying five developers performing a maintenance task on a reasonably-sized system that successful developers reinvestigate methods

less frequently and mostly perform structurally guided searches. La Toza et al. [LaToza et al., 2007] studied the relation between a developer's experience and her navigation behavior. In a study with 13 developers they found that more experienced developers reinvestigate less methods.

Our work supports observations gained in these studies. We further investigate the fine-grained navigation behavior of developers exploring source code for change tasks and for activities.

Biometric Sensing. Recently, researchers also used biometric sensors to learn more about developers' cognitive processes when they perform a change task. Amongst biometric sensors, eye trackers have been used most often to better understand how developers read code elements. One of the earliest studies that tracked developers eye gazes was conducted by Crosby and Stelovsky [Crosby and Stelovsky, 1990]. They investigated how developers read an algorithm written in Pascal and in particular whether the developers' experience in programming influences the way they read the algorithm. Uwano et al. [Uwano et al., 2006] found that developers read first the entire code snippet to get an overview of the program. Rodeghero et al. [Rodeghero et al., 2014] measured at which parts within a method developers look at the most. They devised based on their findings an improved code summarization technique. Other researchers explored how identifier styles influence developers' eye gazes [Binkley et al., 2013, Sharif and Maletic, 2010a]. Further works used other biometrics sensors, such as functional magnetic resonance imaging (fMRI), electro-dermal activity (EDA), or electroencephalography (EEG), to study which parts in the developers' brains are activated when developers read source code [Siegmund et al., 2014], to predict the difficulty a developer perceived [Fritz et al., 2014a], to predict a developer's emotions and progress [Müller and Fritz, 2015], or to predict developers' interruptibility [Züger and Fritz, 2015].

Most of these studies, which use biometric sensors to capture developers' cognitive states, focus on very small code snippets. We analyze developers' navigation behavior in a reasonably sized system for real change tasks.

Developers' Activities for Change Tasks. Exploratory studies in the field report on a variety of activities that developers carry out when they explore

source code to perform a change task in a software system. The activities that are reported in these studies vary in their granularity, ranging from abstract and high-level activities, such as understanding and editing code [Minelli et al., 2015, Ko and Myers, 2005, Ko et al., 2006, Singer et al., 1997, LaToza et al., 2006, Amann et al., 2016, Vans et al., 1999, Brooks, 1999], to fine-grained activities, such as the navigation of code dependencies [Ko et al., 2006]. However, very few studies looked into the automatic recognition of such activities that developers engage in to perform a change task. Coman and Sillitti [Coman and Sillitti, 2008] suggested an algorithm which retrospectively detects different tasks within a developer's programming session. They validated their approach on a small system with three participants who worked on up to five change tasks. Zou and Godfrey [Zou and Godfrey, 2012] tested the proposed algorithm in industry and found that a large number of the tasks identified by the algorithm are in fact subparts of a bigger task. Compared to their approach, we aim to identify activity types and switches that are carried out during the work on a given change task.

1.6.2 Modeling Code Elements' Relevancy

Numerous approaches have been explored that determine the relevancy of code elements. Knowing the relevancy of code elements for a given change task enables, for example, approaches helping to better resume tasks [Kersten and Murphy, 2006], approaches that help developers to switch less between applications by recommending relevant Stack Overflow posts or further relevant artifacts directly in the IDE [Ponzanelli et al., 2014, Čubranić and Murphy, 2003], or approaches that recommend parts in the source code to explore next [Ying et al., 2004, Zimmermann et al., 2004, Robillard, 2005, Buckner et al., 2005, Hill et al., 2007, Piorkowski et al., 2012, Singer et al., 2005]. These approaches differ in the way to compute a code element's relevancy for a given change task. There are approaches which use the system's structure to infer relevancy information from [Robillard, 2005, Buckner et al., 2005] and there are approaches which further use lexical similarities between code elements as indicators for relevancy [Hill et al., 2007, Piorkowski et al., 2012]. Other approaches use historical information sources, such as a system's change history [Ying et al., 2004, Zimmermann et al.,

2004] or interaction logs of either developers who previously solved change tasks in the system [DeLine et al., 2005b, DeLine et al., 2005a, Singer et al., 2005] or of the developer who is working currently on the change task [Parnin and Gorg, 2006, Kersten and Murphy, 2005].

In our research, we build upon these previously explored models. In particular, we investigate whether we can use the fine-grained eye gaze data of developers and the information about the activities developers worked on to improve the recognition of relevant source code elements on different granularity levels.

1.6.3 Online Controlled Experiments

Software developers have always conducted experiments with their software systems. With the increasing rate of releasing software, these experiments can be conducted drastically faster and product teams have the possibility to continuously learn about the users of their product [Bosch, 2012]. Works in the context of online controlled experiments can broadly be categorized into case studies that investigated the challenges with which organizations have to cope when running experiments with their code changes, works that suggest a model for controlled experiments, and works that investigate optimizations of the experimental procedures used. Researchers identified that the culture within an organization is among the biggest challenges that organizations face when experimenting with code changes [Davenport, 2009, Kohavi et al., 2009a, Kohavi et al., 2013, Lindgren and Münch, 2015]. Slow development cycles, proper product instrumentation, and the identification of metrics to test an experiment's hypothesis, and the capturing of user data, have been observed as further challenges [Lindgren and Münch, 2015, Rissanen and Münch, 2015]. Fagerholm et al, [Fagerholm et al., 2014,Fagerholm et al., 2017] explore a model of experimentation, called RIGHT. In particular, they structure the experimentation process into build-measure-learn blocks and found that experiments and product strategies influence one another. However, they observed that the vision of the product remained unchanged. Deng [Deng et al., 2013] explored the trade-off and optimization between the experiment exposure duration and the amount of users which are exposed to an experiment.

Different to these works, we empirically analyze experiments that have been conducted within a mature and large-scale system over the past 2.5 years. In particular, we strive to better understand the development efforts that are carried out for an experiment, as well as to learn the time spans that are required for conducting an experiment.

1.7 Summary and Contributions

The findings gained from our series of exploratory studies and data analyses support our hypothesis H1 that a model based on developers' fine-grained navigation behavior improves the automatic determination of relevant code elements in terms of precision and recall and that such a model enables to automatically recognize developers' activities that are carried out when performing a change task. Furthermore, the results from the data analysis 2 (see Table 1.1) support our hypothesis H2 that a model based on an empirical analysis of experiments and their development efforts can identify bottlenecks in the development process and can identify differences between code changes that are appreciated by users and the code changes that have not been appreciated by users. Our work makes the following contributions:

- we present empirical observations from lab and field studies about developers' fine-grained navigation behavior when performing change tasks and we infer and discuss design recommendations and opportunities for tool support;

- we present types and characteristics of activities that are carried out by developers who perform a change task and demonstrate that we can automatically detect an activity's type and boundary with high accuracy;

- we present a model based on developers' fine-grained navigation behavior that improves the precision and recall upon existing approaches in recognizing task-relevant and task-irrelevant code elements; and

- we present a model of an experimentation process and a large-scale empirical data analysis on 21,220 experiments and their development efforts that have been conducted within a big and mature software system.

1.8 Thesis Roadmap

In the following chapters of these thesis, we answer the research questions that we discussed in Chapter 1.1. All Chapters are based on peer-reviewed publications, for an overview see Figure 1.2.

Chapter 2. This chapter is based on a scientific publication published at 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2014). This work was done in collaboration with my supervisor Prof. Thomas Fritz, David C. Shepherd and Will Snipes, both from ABB Corporate Research, and Christoph Bräunlich, a former student from the University of Zurich. My contributions in this chapter comprise a partial analysis of the data that was captured in the exploratory study, the empirical analysis, and a part of the paper writing.

Chapter 3. This chapter is based on a scientific publication accepted for publication at the 33rd IEEE International Conference on Software Maintenance and Evolution (ICSME 2017). This work was done in collaboration with my supervisor, Prof. Thomas Fritz. My contributions in this chapter comprise the design and execution of the exploratory study and the provision of the technical support thereof, the data analysis of the captured data, and the paper writing.

Chapter 4. This chapter is based on a scientific publication published at 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2015) and an extended journal article thereof that is published in the Journal of Systems and Software. The work for both publications was done in collaboration with Braden M. Walters, Timothy R. Shaffer, and Prof. Bonita Sharif, all affiliated with Youngstown State University. Furthermore, David C. Shepherd from ABB Corporate Research and my supervisor Prof. Thomas Fritz contributed to this work. My contributions in this chapter comprise the design and help

during the execution of the exploratory study and the provision of technical support thereof, the data analysis of the captured data, and the paper writing.

Chapter 5. This chapter is based on a scientific publication at the ACM Student Research Competition at the 39th International Conference on Software Engineering. My contributions in this chapter comprise the design and execution of the exploratory study and the provision of the technical support thereof, the data analysis of the captured data, and the paper writing.

Chapter 6. This chapter is based on a scientific publication at the 39th International Conference on Software Engineering, Software Engineering in Practice track (ICSE SEIP 2017). This work was done in collaboration with Brendan Murphy from Microsoft Research, Laurie Williams from North Carolina State University, and Jennifer Beckmann from Microsoft. My contributions in this chapter comprise the processing of the data, the data analysis, and the paper writing.

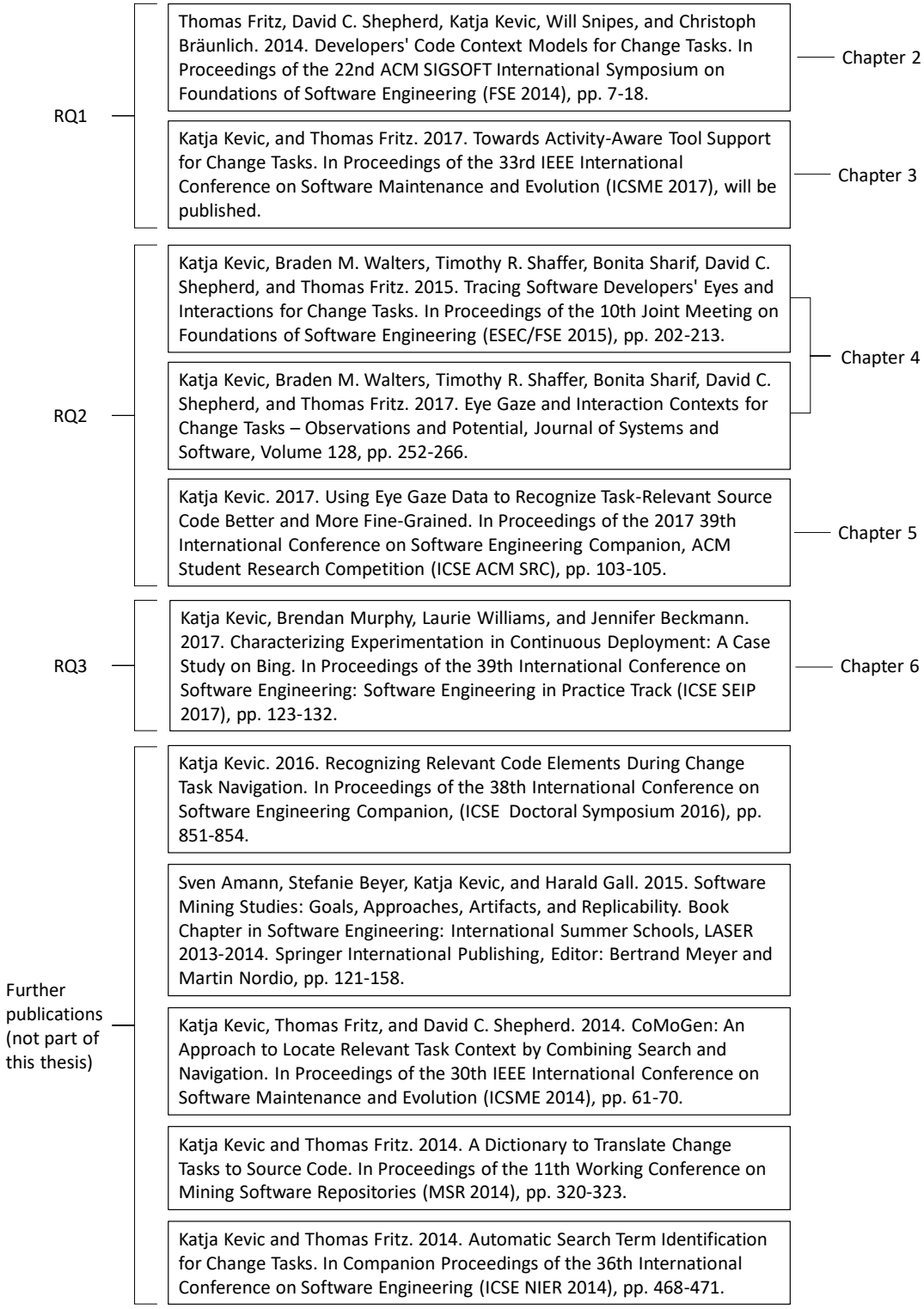


Figure 1.2: The roadmap of this thesis.

2

Developers' Code Context Models for Change Tasks

Thomas Fritz, David C. Shepherd, Katja Kevic,
Will Snipes and Christoph Bräunlich
Published at the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2014
Contribution: Data analysis, and paper writing

Abstract

To complete a change task, software developers spend a substantial amount of time navigating code to understand the relevant parts. During this investigation phase, they implicitly build context models of the elements and relations that are relevant to the task. Through an exploratory study with twelve developers completing change tasks in three open source systems, we identified important

characteristics of these context models and how they are created. In a second empirical analysis, we further examined our findings on data collected from eighty developers working on a variety of change tasks on open and closed source projects. Our studies uncovered, amongst other results, that code context models are highly connected, structurally and lexically, that developers start tasks using a combination of search and navigation and that code navigation varies substantially across developers. Based on these findings we identify and discuss design requirements to better support developers in the initial creation of code context models. We believe this work represents a substantial step in better understanding developers' code navigation and providing better tool support that will reduce time and effort needed for change tasks.

2.1 Introduction

Software developers spend substantial time searching and navigating through code to understand relevant parts of a system for a particular change task [Ko et al., 2006, Singer et al., 1997]. During this process of understanding and then changing code, developers implicitly build *code context models* that consist of the relevant code elements and the relations between these elements, often more generally referred to as task context. Since these models mainly stay implicit in developers' heads and are not persistent, developers have to continuously spend a significant amount of their time creating context models for newly assigned change tasks from scratch [LaToza et al., 2006].

Researchers have suggested that more explicit context models for change tasks[1] can be used to support developers in their work [Murphy et al., 2005]. To form these explicit context models, existing approaches have used methods ranging from a developer manually specifying the context (*e.g.*,, [Robillard and Murphy, 2002]) to automatically inferring the context from a developer's interaction with a development environment (*e.g.*,, [Kersten and Murphy, 2006]). While these approaches have been shown to support developers with change tasks, little is understood about the implicit code context models that developers

[1] We use the term change task to refer to both modification task and bug.

build and their characteristics. With a better understanding of the characteristics of developers' code context models, we might be able to help developers in the initial creation of these models for change tasks, saving time and effort.

To investigate the characteristics that code context models exhibit for different change tasks, we conducted an exploratory study with twelve developers on three change tasks in open source projects. To validate our observations on a broader population of developers and tasks, we conducted a second empirical analysis using data sets from several hundreds of change tasks and eighty developers working on open and closed source projects. Amongst other results, our studies show that developers' context models are highly connected, structurally and lexically, that the code navigation can differ substantially by individual even for the same change task, and that developers start change tasks using a combination of search and navigation and then frequently revisit code elements. Based on our findings we infer design requirements to support developers in the creation of code context models and discuss the design of such an approach.

This paper makes the following research contributions:

- It identifies important observations on the characteristics of code context models based on an exploratory study with 12 developers on three open source projects.

- It provides an empirical analysis of the findings on data collected from eighty developers working on a variety of change tasks on open and closed source projects.

- It identifies design requirements and discusses the design of an approach to support developers in the creation of code context models.

This work represents a substantial step in better understanding developers' code navigation and implicit context models for change tasks that will help to provide better tool support with the potential for real-world impact on the development process, in particular on reducing the time and effort required for change tasks.

2.2 Exploratory Study

In the process of performing a change task, developers build up *code context models*[2]—code elements and relationships between these elements that are *relevant* to the change task. In this study, we investigate these code context models based on three specific concepts of relevance: (a) relevance as perceived by the developer, (b) relevance as defined by the actual code change and (c) relevance as defined by the explicit navigation activity of the developer. Since our ultimate goal is to support code context creation we also investigated developers' code navigation tendencies to understand how code context models could be created. In particular, we wanted to address the following questions:

(1) What are common characteristics that code context models exhibit?

(2) How do code context models vary based on different definitions of relevance?

(3) How do developers' code context models and navigation behavior vary for different tasks?

To investigate these questions, we conducted an exploratory study with a blocked subject-project study setup [Basili et al., 1986] with twelve software developers. Each developer worked on one of three different change tasks for open source projects.

2.2.1 Study Method

For this experiment we chose three open source systems in Java that all have an open task repository, recent development activity and a code base big enough to preclude a systematic understanding of the entire system. Specifically, we chose FreeMind [FreeMind, 2017], Java PasswordSafe (JPass) [JavaPasswordSafe, 2017] and Rachota [Rachota, 2017]. For each system we chose one open change task that two of the authors were able to perform in less than one hour. Furthermore, we chose tasks for which the change could be observed in the graphical user

[2]Murphy *et al.* ([Murphy et al., 2005]) introduce the broader term *task context* that extends our notion to arbitrary artifacts.

interface. All three change tasks were reported as bugs, however, the JPass and FreeMind tasks could have been categorized as enhancement or modification task. Thus, we will mostly refer to all three tasks with the more general term *change task*.

Each developer who signed up for the study was randomly assigned to one of the three tasks. We then provided each participant a document with instructions and access to a virtual machine that was set up with an Eclipse IDE[3] and a workspace that contained the assigned change task description and project. We decided to set up a virtual machine for each participant on the Amazon Elastic Compute Cloud [AmazonWebServices, 2017] to allow for remote and independent access using his own computer setup and thus to affect the "normal" behavior as little as possible. The participants were instructed to first run the application and observe the current behavior to be changed before looking at the code and trying to perform the change. Furthermore, the instructions told the participants to answer a set of questions after either completing the change task successfully or after 75 minutes to limit the total time required of a participant to 90 minutes—75 minutes for the change task and 15 minutes for the questions.

In the questions, the participants were asked to sketch a model of the source code elements, such as classes, methods and fields, and the relationships they considered relevant for understanding and making the change. For the sketch, the participants were allowed to use pen and paper or their favorite drawing tool and they were encouraged to use any notation or form they wanted to. The rest of the questions in the set addressed the experience of the participants.

To make sure that the tasks are solvable in the given time and the questions are understandable by the participants, we conducted pilot studies with three graduate students, each performing one of the three tasks. The pilots confirmed our assumption on the timing and we only slightly altered the question on the model sketching part to explicitly state that developers are allowed to use pen and paper for the sketch.

[3]Eclipse IDE for Java Developers, version 3.7, `eclipse.org`

2.2.2 Subjects

We recruited subjects through email and personal contact. To be eligible, subjects had to have experience programming in Java. We ended up with 12 participants that we randomly assigned to one of the three tasks, four for the FreeMind task (F1-F4), four for the Java PasswordSafe task (J1-J4) and four for the Rachota task (R1-R4). Of these twelve developers, five worked in a company, four were graduate students and three faculty members in Computer Science, all with a background in software engineering. The subjects' programming experience ranged from 8 to 16 years (average of 11.8) with between 0 to 12 years (average of 4.5) of professional programming experience. For each task we made sure to have at least two developers with professional development experience and one graduate student to report on. Two of the subjects were female, ten male. On a five point Likert-scale with 1 (strongly disagree) to 5 (strongly agree), all subjects agreed or strongly agreed that Java is one of their primary programming languages (average of 4.75), and were very familiar with the Eclipse IDE (average of 4.17).

2.2.3 Projects and Change Tasks

FreeMind. The FreeMind project (version 0.9.0 RC 15) is an open source mind map editor consisting of 52.5k non-commented lines of code (NCLOC), 439 top level classes, and 45 packages. We selected a task for this project (ID 3420227, [FreeMindBug, 2017]) that was still open and observable. This change task addressed FreeMind's failure to save a map after an encrypted node was added as well as the inadequate notification upon failure. We limited the scope of this potentially large change by asking the subjects to add a reasonable explanation to the "Save Failed" dialog. This change required users to propagate exception information from the `save` method of `EncryptedMindMapNode` to the user action (`actionPerformed` in `SaveAction`) and finally to display the improved message to the user. The call chain between `actionPerformed` and `save` is relatively long (11 method calls in total) and can be challenging to follow. Fortunately, when reproducing the failure, which we asked all subjects to do before making

the change, a stack trace was printed to the console that contained the relevant call chain.

Java PasswordSafe. The Java PasswordSafe (JPass) project (version 0.8 final) is an open source password management system consisting of 13.5k NCLOC, 167 top level classes, and 18 packages. We again selected a change task (ID 2933526, [JavaPasswordSafeBug, 2017]) that was still open at the time and observable. This task addressed the lost selection and undesired scrolling that occurs when the application is unlocked after coming out of the sleep state. For this change task, we expected subjects to save the selection index in order to reselect and center the appropriate item after the application was unlocked. While classes `UnlockDbAction`, `LockDbAction`, and `PasswordSafeJFace` were all relevant for this task, only one to two methods in `PasswordSafeJFace` needed to be changed. During this task, subjects familiar with the Standard Widget Toolkit [EclipseSWT, 2017] may have benefitted, although prior knowledge was not necessary. This application also made extensive use of console logging. Observant developers could use these log messages as a starting point for searches.

Rachota. The Rachota project (version 2.4) is an open source time tracking utility where users can track the time spent on each task. It consists of 18k NCLOC, 53 top level classes, and three packages. We selected a task (ID 2658881, [RachotaBug, 2017]) that was open and observable. This task addressed the problem of newly created tasks failing to show in the 'History' tab. For this task, we expected subjects to trigger an update of the History tab's underlying model upon task creation. Three classes that are relevant to the task are extremely large (`HistoryView` has 1800 NCLOC with 42 methods, `MainWindow` has 1125 NCLOC with 28 methods and `DayView` has 1807 NCLOC with 50 methods). These large classes, along with the fact that the application offered no logging or relevant stack trace made it more difficult for users to find a starting point in the code base.

2.2.4 Data Collection and Analysis

We used a combination of qualitative and quantitative methods motivated by the ones described by Seaman [Seaman, 1999]. We used participant observation by recording each participant's screen and having access to their actual workspace after the session, in addition to asking the participant a set of questions. From the participants' sessions we collected three types of data: patches for the successful completion of the change task, videos capturing the developers' screens during their work on the task and the artifacts that contained the answers to the questions, including the sketched models. To record a developer's screen, we automatically started a screen recording application at the beginning of a developer's session. For the questions, we asked developers to send us their answers by email after they finished. We transcribed and coded the patches, the screen recordings and the collected answers. The transcripts together with further study artifacts are available at [StudyArtifactsCoMoGen, 2017].

From the videos we determined the time that each participant took to complete a task. We chose the point at which a participant validated the correctness of his change in the user interface of the application as the finish time. Even though the instructions stated that participants should move onto the questions part after 75 minutes to limit the total amount of effort spent, three participants chose to continue. Two of these three participants, J4 and R4, did not succeed in performing the appropriate change and at some point stopped working on it. Both participants closed Eclipse at the end which we used as the finish time. Table 2.2 presents the time participants took to complete the task or until they stopped.

For investigating and comparing the three different code context models, we determined the source code elements and relations in these models from the data collected.

Developer Models. For the code context models based on the developer's relevance definition, which we refer to as *developer models* in the following,

[4]This was only used in the FreeMind task and opened up the parent class.

Table 2.1: Developer navigation steps transcribed from the screen-captured videos (several of these refer to tool support provided in the Eclipse IDE).

Structured Navigation Steps	
navigation aids	call hierarchy, type hierarchy, find references
debugger	step into, step return, stacktrace click
editor	quick documentation, open declaration, quick fix[4]
Unstructured Navigation Steps	
package explorer	expand item, open item
search	Java, file, find in file, outline view
editor working set	back, forward, open from editor tab
editor	scan

we coded the models sketched by the developers. We acknowledge that these sketches may not be a complete or accurate representation of developers' implicit context models, thus necessitating the use of complementary models, in particular the code navigation model. We believe, however, that these sketches encode important or prominent features of these implicit models of which developers are conscious. For each sketch, we determined the code elements the sketches explicitly referred to. Since all twelve models contained references to classes but four did not contain any methods and three did not contain any fields, we only examine the classes used in these models for a fair comparison in the following.

Patch Models. For the code context models from the actual patch, which we refer to as *patch models*, we determined the classes and methods that were changed as well as the types that were used and the methods that were called in the actual change.

Code Navigation Models. For the models defined by the developer's explicit navigation behavior, which we refer to as *code navigation models*, we transcribed the screen recordings and coded the resulting transcripts. Since we are interested in the navigation of a developer through the program code, in particular the

classes and methods, we transcribed the structured and unstructured navigation steps a developer took. We considered a navigation structured if a developer explicitly navigated from a code element A to a code element B along a structural relation using some tool support in the IDE, where code elements were defined as classes, methods or fields and structural relations referred to call, implements and usage relations. Table 2.1 presents a summary of all transcribed navigation steps. For each step we recorded the step, the target element and its type as well as, in case of a structured step, the source element, its type and the relation followed.

The navigation steps that we transcribed do not explicitly capture the code editing by a developer. However, since we think it is reasonable to assume that a developer has an understanding of the elements he uses or calls in his code change, we added these elements to the code navigation model if they were not yet in it, which was rarely the case.

For transcribing code navigation one has to determine which code elements a developer is examining at any point in time, which is challenging as described by [Robillard et al., 2004]. While transcribing the video, we used the mouse pointer as a clue and examined the actual code base to determine which method a developer was in. We also used the keyboard events to determine the tool support a developer used. Observational studies are subject to observer bias which may lead to omitting navigation instances or characterizing them incorrectly. To mitigate this risk, we had an initial phase in which three investigators transcribed one video and cross-validated the results to make sure no major differences occurred. After this phase, one investigator transcribed all videos and random samples were picked for cross-examination by the other two investigators revealing no major differences.

2.2.5 Study Results

Based on the analysis of the qualitative and quantitative data we gathered, we made several key observations with respect to the three questions we set out to study. Given the exploratory nature of our study, we will discuss these observations and their implications mainly alongside the presentation of descrip-

Table 2.2: Summary of descriptive statistics on participants' background and exploratory study ($pro = professional$, $grad = graduate\ student$, $fac = faculty$, ✓ $= success$, ■ $= failure$, $Cl = classes$, $Me = methods$, $Deb = debugging$).

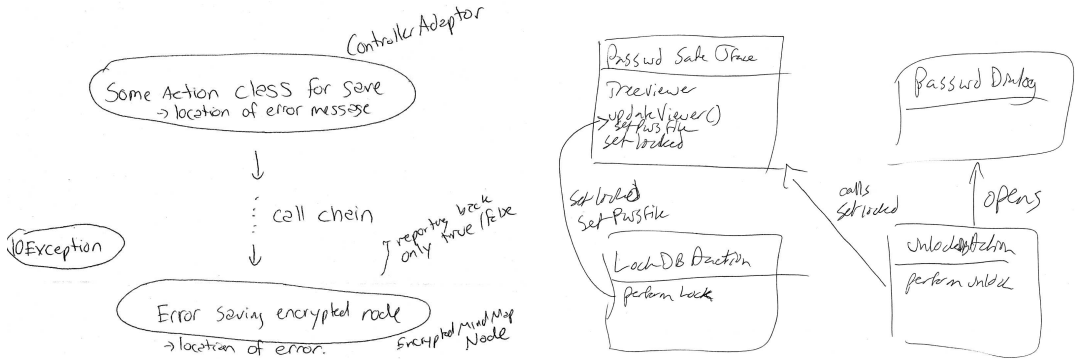
Project	**Freemind**				**JPass**				**Rachota**			
ID	**F1**	**F2**	**F3**	**F4**	**J1**	**J2**	**J3**	**J4**	**R1**	**R2**	**R3**	**R4**
Job	pro	pro	grad	grad	pro	pro	pro	fac	fac	grad	pro	fac
Years Pr. Exp	10	11	8	9	12	12	11	12	15	11	15	16
Time (min)	39.7	22.0	59.7	70.9	8.6	64.5	62.0	101.1	53.8	100.6	36.6	114.4
Success	✓	✓	✓	✓	✓	✓	✓	■	✓	✓	✓	■
Dev.Model												
Cl	4	4	5	4	4	4	5	4	6	4	6	9
Patch Model												
Cl	16	15	11	16	2	6	3	-	5	1	1	-
Me	25	23	18	23	2	11	7	-	10	2	2	-
Nav. Model												
Cl	37	16	19	22	6	19	12	19	20	13	10	16
Me	42	25	36	38	12	28	27	32	31	78	22	35
Nav. Steps												
All	116	106	341	177	67	408	140	570	349	553	326	282
Structured	101	57	229	41	46	279	64	459	101	42	130	76
Revisits	47	40	250	112	30	305	88	453	248	396	241	155
Deb	26	54	219	11	33	250	17	441	78	39	160	6

tive statistics. In the presentation of the observations we only included the successful subjects (ten of the twelve subjects) since we did not have patches for the unsuccessful subjects. A summary of the statistics gathered is presented in Table 2.2.

O1—Developer models are small, abstract and highly connected.

Across all subjects and tasks, developer models are consistently small with a mean (M) of 4.6 class elements (standard deviation SD of 0.8). Even though the size of patch models showed big variances for different tasks ($M = 14.5$ classes for FreeMind, $M = 3.7$ for JPass and $M = 2.3$ for Rachota) and the code navigation models varied widely across all subjects on the class level (overall $SD = 8.5$ with $M = 17.4$), the size of the developer models remained consistently small.

Developers generally used abstraction in their models. Instead of using concrete class names developers recorded the concepts or functionality they were interested in, *e.g.*,, subject F2 used "some action class for save, location of error



(a) F2's developer model (b) J1's developer model

Figure 2.1: Developer models for FreeMind and JPass.

message" to denote the class `ControllerAdaptor` (see Figure 2.1a) and J2 stated "Main View" and put the actual class name in brackets close by. However, the level of abstraction used in the models varied by subject and task. For instance, all four FreeMind subjects used a very high level of abstraction, whereas subjects on the JPass task included more detail in their models. An example to illustrate this difference is shown in Figure 2.1; Figure 2.1a shows F2's developer model for the FreeMind task which is highly conceptual, abstracting from direct call relations to transitive call chains, and Figure 2.1b shows the model of developer J1 on the JPass task which resembles a class diagram with details of the code.

All developer models were also highly connected. In fact, all models were fully connected at class-level excluding one class element in subject F2's otherwise connected model. On average, there were 5.3 relations in a developer model ($SD = 2.1$) and these relations mainly referred to method calls, but also to contains and inheritance relations.

> ***O2—Patch model size has little influence on the size of code navigation models.***

For the three tasks we investigated, the average number of methods in the patch model had almost no influence on the number of methods in the code navigation model. The patch models for the FreeMind task were the largest and the most scattered, containing an average of 22.2 methods ($SD = 3.0$) over 14.5 classes.

The patch models for the JPass task and the Rachota task were both much smaller ($M = 6.7$, $SD = 4.5$ and $M = 4.7$, $SD = 4.6$ respectively) as well as less scattered (3.7 and 2.3 classes). In spite of the bigger patch models, the code navigation models for FreeMind were smaller than the ones for Rachota, with an average of 35.2 methods ($SD = 7.3$) in the FreeMind models and 43.7 ($SD = 30.1$) for Rachota. A similar lack of correlation is seen when, in spite of patch models of roughly equal size, JPass's code navigation models were on average a lot smaller ($M = 22.3$, $SD = 8.8$) than Rachota's. This can also be seen in the Pearson's correlation coefficient between the patch and the code navigation model size being close to zero overall with $r = .006$. This observation implies that *a bigger and more scattered change does not result in a developer navigating through more method elements to make the change.*

> ***O3—Even for concise and successful changes, code navigation models can differ substantially on class as well as method level.***

Code navigation models can vary substantially across developers, even for tasks that require only small changes. For example, while there was some agreement on four core classes for the JPass task, *i.e.*,, all four classes were in all navigation models and three of these four were in all developer models, there was a wide variance outside of these four classes. The three subjects had between 6 and 19 classes in their navigation models with an average overlap of elements with at least one other subject's model of only 52.6%. On method level, the variance was larger as models ranged from 12 to 28 elements with only 3 methods that all three subjects had in common. In class `PasswordSafeJFace`, one of the core classes for this change task, the three subjects inspected 21 different methods but only one of these 21 methods was inspected by all subjects.

> ***O4—Code Navigation Models are highly connected (structural cohesion).***

Upon inspecting all code navigation models for all successfully completed change tasks we found that, on average, 73% of the class elements in a code navigation

model are connected with at least one other class through a call, usage or implements relation. Six of the ten code navigation models centered around one large connected group of classes and zero or more additional classes with no connections. By cross referencing these unconnected classes with the transcripts we observed that the unconnected elements were often visited towards the beginning of the task using unstructured navigation steps, prior to developers finding a point of reference to start a deeper, more structured investigation. For example, for the code navigation models of the three subjects on the JPass task, there are nine classes that are not connected to more than one other class and seven of these nine were navigated to within the first few steps. Figure 2.2 illustrates an example of a code navigation model for JPass, including a numbering to show the order in which elements were navigated to. In this example, most elements are connected except for some that the developer navigated to in the beginning and two elements from seemingly random selections later on (30 and 31).

O5—Navigation Sequences are largely determined by lexical similarities (lexical cohesion).

In our transcripts we observed that developers often subsequently visited code elements that share identifying terms within their identifiers, in particular in the beginning of the investigation. Upon inspecting all subsequent visits, either from one class to another, one method to another one within the same class, or one method to a method in another class, we found that over all subjects and tasks, 43% (±17%) of the elements visited subsequently are lexically similar. In this paper, we define two identifiers as lexically similar if, after splitting up identifiers according to camelCase notation, they have at least one term in common. This result supports the observations made by other researchers (e.g. [Lawrance et al., 2008, Ko et al., 2006]). We also found that developers pay overall more attention to lexical similarities when they switch from a method to a method in another class. On average 69% of the method switches to a different class are lexically similar, whereas only 29% of the method switches within the same class are lexically similar.

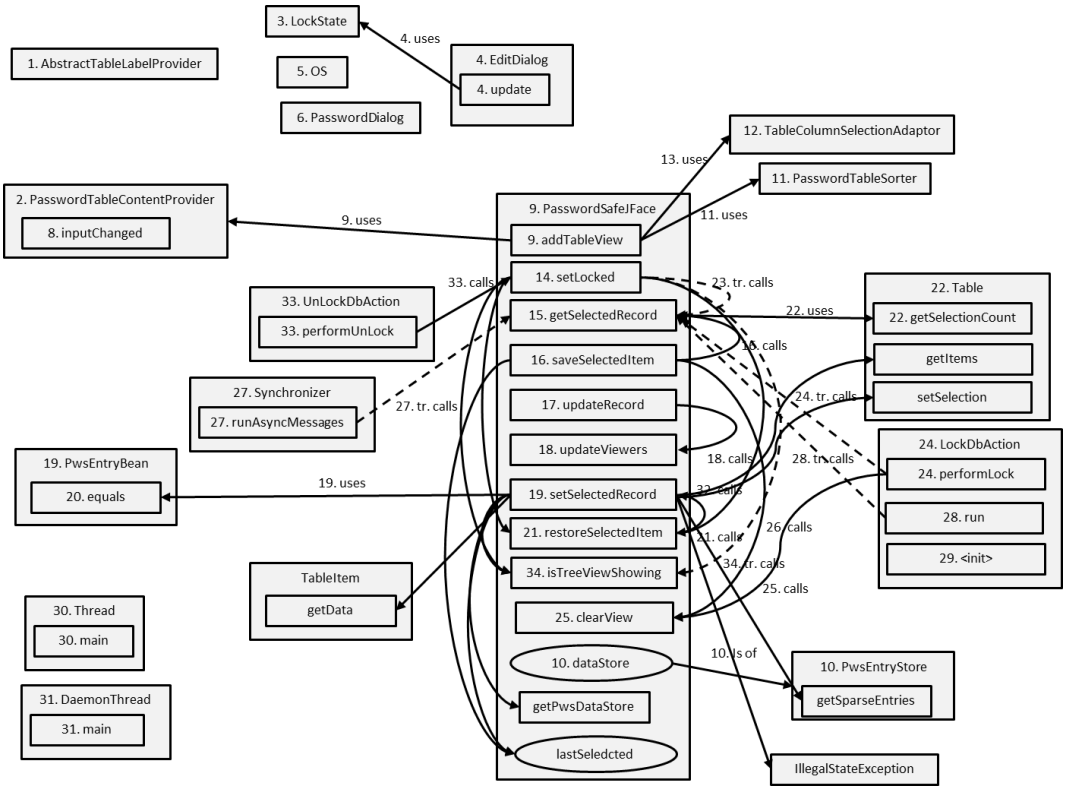


Figure 2.2: Code navigation model for subject J2.

O6—Developers start with a combination of search and structural navigation.

In a study on a small project with 500 lines of code, Ko *et al.* [Ko et al., 2006] found that developers first search for information, then engage with the information to decide whether it is worth continuing by navigating the relationships between information, before finally editing the code. Similarly, Sillito *et al.* [Sillito et al., 2006] identified that developers exhibit a behavior of 'finding initial focus points' and then 'building on those points' through navigation and exploration. Our exploratory study on the three projects corroborates these initial findings. Of the twelve subjects, nine performed an explicit search within the first 8 steps, and the other three found an initial starting point in the code by scanning the package structure rather than explicitly searching. Seven of the nine subjects that performed searches found starting points from the search. In these cases,

within the next 10 steps they spent an average of 4.1 steps following call and execution relations from one of the search results and an average of 4.3 steps scanning one of the search results. More qualitatively, from the code navigation models generated, one can see that developers explored call, declaration and execution relationships a couple of steps out from search results, often revisiting the results and the intermediate elements once determining the relevancy of the element. This suggests that developers start their tasks with a combination of a global search for information and then navigate the structural relations, in particular call relations, to comprehend more about the context of the elements.

> ***O7—Developers frequently revisit code and take less time if their navigation is more structured.***

In their navigation, developers revisit elements more often than they navigate to new code elements. Over all subjects and tasks and including debugging steps, 68% of all navigation steps were revisits, with a mean revisit rate of 61.4% (SD = 14.7%) per subject. Not surprisingly, the more revisit steps a developer performed in his navigation, the more time he spent on the whole change task (Pearson's $r = .72$). Furthermore, the higher the ratio of structured versus unstructured navigation was, the less time a developer spend on the change task (Pearson's $r = -.49$). This result supports the observation that Robillard *et al.* [Robillard et al., 2004] made on successful developers performing more structurally guided searches than unsuccessful ones, only that we look at the time of completing a change task rather than success.

2.3 Empirical Analysis

To further investigate the characteristics and variations of code context models and validate the findings of our exploratory study, we conducted an empirical analysis of two data sets from several hundreds of change tasks and eighty developers working on open and closed source projects.

2.3.1 Data Sets

We used two data sets, denoted as Blaze and Mylyn data, for the empirical analysis. Both data sets capture developers' interactions, such as selects and edits, with an IDE. We used these two data sets for the different aspects they capture about a developers' work within an IDE, the different populations and IDEs they capture and their availability. While the Mylyn data contains information on the specific tasks developers worked on and the changes they committed for resolving these tasks, the Blaze data contains information on search instances and the exact order of events.

Mylyn Data. The Mylyn data consists of change tasks, task contexts capturing interaction data for a change task, and patches of the Mylyn project. We chose to analyze the Mylyn project as it is a reasonable-sized project (466k NCLOC) that provides information on developers' interaction with the Eclipse IDE for a reasonable number of change tasks. The interactions are thereby captured using the Eclipse Mylyn project [Mylyn, 2015, Kersten and Murphy, 2006]. From the Bugzilla [Bugzilla, 2017] repository for the Mylyn project, we retrieved 9920 change tasks reported between 07/18/2007 and 02/20/2014. From this set, we filtered all change tasks that did not have a patch and a task context associated, resulting in a total of 2253 change tasks. For each change task, we extracted several features, such as the severity of the change task and the comments stored within the change task. By linking the change task ID to the commit comments in the change history of the Mylyn project, we identified the patches for each change task and extracted the changed classes and methods using ChangeDistiller [Fluri et al., 2007]. In addition, for each change task we retrieved the associated task context and extracted the classes and methods a developer selected in the process of performing the change task.

For each of the 2253 change tasks in our Mylyn data, developers changed on average 5.7 ($SD = 16.2$) classes and 13.0 ($SD = 69.2$) methods and selected an average of 47.2 ($SD = 139.7$) classes and 18.8 ($SD = 31.1$) methods. The change tasks in this data set are categorized into different task types based on the severity field:

915 normal, 719 enhancement, 313 minor, 130 trivial, 127 major, 30 critical and 19 blocker. The time period of a task context, which denotes the time from the first captured code selection for the task to the last, varies a lot with an average of 18.5 days (SD = 87.3). Thus, it differs substantially from the short change tasks investigated in our exploratory study and most other similar empirical studies in related work. Overall, there were 31 developers, each working on at least one of the 2253 change tasks.

Blaze Data. Blaze is a Visual Studio extension that logs interaction data, in particular all actions a developer performs within the IDE or that are executed by Visual Studio itself [Snipes et al., 2014]. Blaze acts as a global event handler, listening for all GUI events within Visual Studio. For each event, Blaze records the key attributes, such as the name and type of the event. If existent, Blaze also records the file name and the currently selected line number. All events are registered along with an anonymized unique identifier for each Blaze user that allows to investigate differences between developers.

The Blaze data used in our analysis contains data recorded from 59 developers and over 8000 hours of development activity. It was collected from developers in ABB's globally distributed industrial software development community that volunteered to share their data.

2.3.2 Data Analysis and Results

To examine the characteristics of code context models and validate the findings of our exploratory study, we performed a set of analysis over the two data sets.

Mylyn Data. To examine if the size of the patch influences the size of the task context (***O2***), we compared, for each change task, the changed classes and methods of the patches with the selected classes and methods within the task contexts. Figure 2.3 illustrates the high variance and lack of a general trend between the number of changed methods and the number of selected methods which is similar on class level. A Pearson's correlation coefficient between the number of changed and selected elements of r_{class}=.257 (p<.001) on class and of r_{method}=.202 (p<.001) on method level supports this lack of a trend.

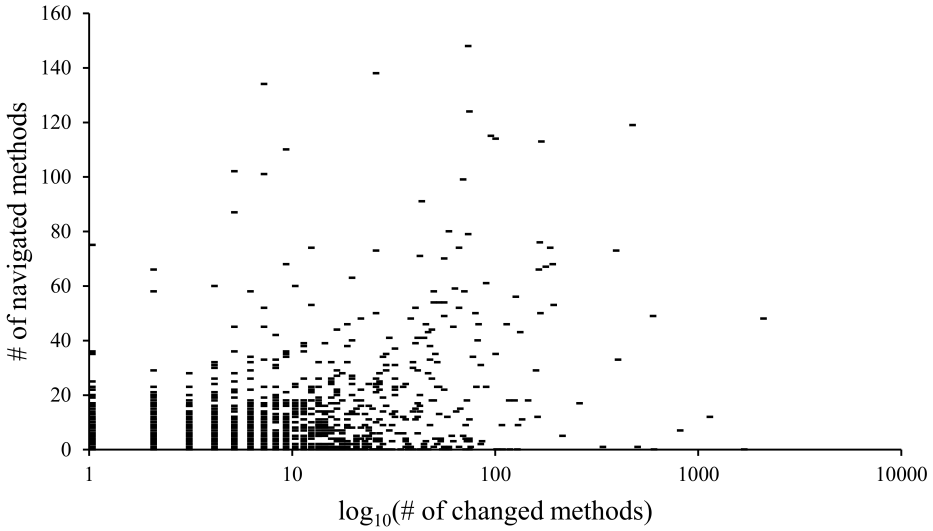


Figure 2.3: Number of selected methods plotted against number of changed methods.

Both correlation coefficients are of small effect sizes (r<.3). The coefficient of determination, R^2_{class}=.066 for r_{class}, and R^2_{method}=.041 for r_{method}, measures the amount of the variability in the number of changed classes, respectively methods, that is shared by the number of selected classes, respectively methods. R^2_{class} implies, that the number of changed classes and the number of selected classes share only 6.6% of variability and that 93.4% cannot be explained. On method level, only 4.1% of the variability is shared, denoting that 95.9% of the variability cannot be explained. These results support ***O2***.

O8—Type and discussion length of a change task can significantly influence code navigation models.

Since our results show that the size of a patch has little influence on the navigation behavior, we investigated the influence of other aspects of a change task. In particular, we looked at the type and the discussion length of a change task, since these might be reflective of the complexity of a change task, assuming that complicated or unclear concepts often require longer discussion threads. Therefore, we examined if we can predict the task context size as small or big

on the basis of a change task type using multinomial logistic regression. We defined a task context as small if it contains less than the median size of all task contexts (8 different classes and 3 different methods) and big otherwise. A chi-square test results in *chi square*= 40.624 (p<.000 with df=6), showing that the predictive power significantly increases when the change task type is added to the model. Looking at the parameter estimates for all seven change task types allows us to interpret effects by comparing two change task types against each other. For example, the odds of having a small task context for a "trival" change task are 2.34 times higher than for an "enhancement" (with p<.001). When comparing all pairs of types, significant effect sizes exists for enhancement and trivial, enhancement and minor, enhancement and major, and enhancement and normal, showing that a change task of type enhancement usually results in a bigger task context. Our comparison of the number of comments within a change task and the navigation behavior, represented by the number of selected elements, shows that there is a correlation with a medium effect size with a Pearson's r=.30, p<.001 on class and Pearson's r=.38, p<0.001 on method level. At the same time, the number of comments and the patch size (number of changed elements) are only correlated with a small effect size with Pearson's r=.11, p<.001 on class and Pearson's r=.27, p<.001 on method level.

We also used the Mylyn data to further examine the observation that even for concise changes code navigation model can differ substantially (***O3***). Since in the history of a project, there are usually no two change tasks that are exactly the same and no two developer complete the same task, as we had it in our exploratory study, we used an approximation for concise changes. In particular, we approximated concise changes as the ones in the Mylyn data that changed at most 2 classes and 3 methods (the median of changed classes, respectively methods over all change tasks). After extracting these changes from the Mylyn data, we retrieved all pairs of changes that had at least 1 method in common to approximate for similar changes as we had in our exploratory study. Overall, we identified 49 pairs of change tasks that changed at most 2 classes and 3 methods and that had at least 1 method in common in their changes. When comparing the task contexts of each of these pairs, we found substantial differences on class

and method level. Out of the 49 pairs, only 13 pairs have an overlap on class level within their task contexts and out of the 13 pairs, 12 pairs share 1 class element and 1 pair shares 2 class elements within their task contexts. On the method level, out of the 49 pairs, only 7 pairs share exactly one method within their task contexts. Since the task contexts of all change tasks of the 49 pairs have an average of 5.35 ($SD = 4.57$) classes and 5.49 ($SD = 4.67$) methods, the overlap in task contexts is relatively small. While it is impossible to recreate the study in which multiple developers perform the same change task and this analysis only approximates it, our results show that even if the changes of different concise changes overlap, the code navigation models are substantially different, providing some evidence that ***O3*** holds.

For the observation that code navigation models are highly connected (***O4***), we examined whether the elements within a task context are structurally connected. We considered elements as structurally connected if they either belong to the same class or if there is a call relationship between the elements on the method level. Disregarding 69 change tasks with only one element in their task context, we found that on average 68% ($SD = 29\%$) of the task context elements are structurally connected, supporting ***O4***.

For the observation that code navigation models exhibit a high lexical cohesion (***O5***), we examined whether the elements within a task context are lexically related. To determine the lexical cohesion within task contexts, we split the elements according to the camelCase rule (e.g. SetupHelper is split into "Setup" and "Helper"). Resulting stop words, such as "get" or "set" were removed. We then consider an element as lexically connected, if it shares terms with at least 50% of the task context's elements. Disregarding again the change tasks which only contain one element, we found that on average 61% ($SD = 35\%$) of the task context elements are lexically connected, supporting ***O5***.

Blaze Data. Our exploratory study as well as a small study by Ko *et al.* [Ko et al., 2006] show that developers often begin a task by searching for an initial point and then move outwards using structured navigation (***O6***). To examine if this pattern also holds for professional developers in the field, we analyzed the Blaze data that captures interaction data from 59 professional developers,

since the Mylyn data does not contain information on the searches performed. Since the Blaze data does not contain task boundary information, we can not examine whether developers start a task with searches, but we focus our analysis on whether there is an increase in structured navigation right after a search. Therefore, for each of the 59 developers within the Blaze data, we compared the general use of structured navigation over all of a developer's work in the IDE with the structured navigation in the minute immediately following a search. This again is only an approximation, but should provide evidence whether the observation also holds in the field.

We consider the following events as structured navigation in the Blaze data:

- Go To Definition (F12): brings up the code that defines the selected identifier.
- View Call Hierarchy (Ctrl+K Ctrl+T): provides a two way analysis of an identifier's dependencies and uses.
- Class View (Ctrl+W, C): provides a browser and search function for classes and class hierarchy.
- Find All References (Ctrl+K,R): provides a list of lines that reference an identifier.
- Navigate to Event Handler: brings up the event handler for an object in the XAML editor.
- View Class Diagram: generates a class diagram.
- View Object Browser: is a search tool and browser.

Unstructured navigation events include selecting a file in an explorer window, selecting the tab for a file, using arrow and page up/down keys to go up/down through a file, scrolling and clicking on a file element.

To examine whether there is an increase in structured navigation after a search, we define the metric structured navigation events per minute (SNM). For each developer, we counted the number of structured navigation steps overall sessions

recorded and calculated the average SNM per developer ($\text{SNM}_{General}$). In addition, for each developer we identified all searches the developer performed, counted the number of structured navigation steps in the minute after the search and calculated the average SNM over all searches by the developer ($\text{SNM}_{AfterSearch}$). In this analysis, we consider two kinds of search that we treat separately: the execution of the command "Find in Files" and the use of the Sando code search [Sando, 2017]. Since all 59 developers used the command "Find in Files" (FiF), we collected 59 pairs of SNMs ($\text{SNM}_{General}$,$\text{SNM}_{AfterSearch}$) for FiF. Only 36 developers used the Sando search tool in their work. Therefore, we only collected 36 pairs of SNMs for Sando. A Wilcoxon Signed Ranks test showed that for both searches, there is a significant difference in the use of structured navigation immediately after a search compared to the general use of structured navigation with p=.006 (T=442, r=-.25) for FiF and p=.015 (T=166, r=-.29) for Sando, providing further evidence for ***O6***.

To examine if developers frequently revisit code (***O7***), and since the Blaze data does not contain information on when the work on a task started or ended, we performed a processing step to partition the Blaze data into sessions, each of one hour length. We chose one hour since this is a few minutes more than the average time participants used in our exploratory study to complete a change task. Per session, we then counted the number of class visits ($\#ClassesVisited$), i.e. the number of times a developer selected a class different to the one that was selected beforehand, and the number of distinct classes visited per session ($\#DistinctClasses$). The percentage of classes revisited can then be calculated as $\frac{(\#ClassesVisited - \#DistinctClasses)}{\#ClassesVisited}$. Over all participants and sessions, a developer visited an average of 16 classes per hour, with 6 distinct classes, resulting in an average percentage of revisiting of 62.5%, and supporting ***O7***.

2.4 Threats to Validity

Exploratory Study. By applying a blocked subject-project study setup with developers from various backgrounds and three different change tasks of three active open source systems we tried to limit the threats to the external validity of

our exploratory study and the experiment. To study change tasks representative of realistic situations, we used change tasks from active open source systems with a size big enough to preclude systematic understanding of the entire code base. A limitation of our study is that all tasks were solvable in less than two hours and thus might not represent the broad range of tasks that exist. We tried to mitigate this risk by choosing the change tasks as randomly as possible (see Section 2.2). Another threat is the limited size of our subject sample and the small number of change tasks which limits our study's generalizability. We tried to mitigate this risk by cross-sectioning full-time developers and researchers from different companies and universities with multiple years of programming experience.

In our exploratory study we focused on Eclipse and Java since they are amongst the most commonly used IDEs and programming languages. Navigation might differ depending on the tools provided in the IDE and language structure.

By screen capturing the participants we could only tell which elements they selected, but not which ones they looked at. This process misses elements and relations that were not explicitly followed through navigation steps, but our focus was on an obvious set of elements rather than an approximation of everything developers might have looked at. In future studies, we plan to explore the use of eye-trackers to also gather information on where developers look.

Empirical Analysis. To increase the external validity of our observations from the exploratory study, we conducted an empirical analysis on data sets from the field. Since the data sets used do not capture the same kind of data captured in our exploratory study, we used approximations, such as partitioning the Blaze data into one hour sessions rather than task sessions, or using a general metric on structured navigation rather than per task. These approximations pose a threat to the construct validity of our results. We tried to mitigate this risk by using two different data sets that captured different aspects of developer's interaction data to better approximate for the analysis of the observations. Furthermore, we never claim to fully validate our observations, but point out that the empirical analysis provides further evidence strengthening the support for the observations.

For the analysis on the Mylyn data, we were only able to analyze 2253 out of 9920 change tasks of the project, which implies that the observations are only denotative for 23% of the project's change tasks. Since developers manually start and stop the capturing of the task context for a change task, this empirical analysis is also threatened by possibly polluted task contexts. Also, we only take into account call relationships and class affiliation when analyzing structural relations between code elements for ***O4***, and do not include type hierarchy. However, this only results in our result being lower than it could be. Finally, the correlation analyses for ***O2*** and ***O8*** suffer the third-variable problem, which means that we cannot argue about the causality of the correlation.

2.5 Discussion

The exploratory study (Section 2.2) and the empirical analysis (Section 2.3) revealed unique characteristics about the behavior of developers when working on a change task. Table 2.3 summarizes key observations along with inferred design requirements for tool support. While there are several approaches to explicitly or implicitly capture task context, such as Mylyn [Kersten and Murphy, 2006] or CodeBubbles [Bragdon et al., 2010] (see Section 2.6), these approaches are limited in the support of the presented design requirements. In the following, we discuss design considerations for tool support for change tasks that explicitly captures developer's code context model.

2.5.1 A Code Context Model Tool

To support a developer in a change task, a tool should not only try to best depict the developer's current code context model—his representation of the relevant code elements and their relations—, it should also provide proactive and relevant context to the developer in all activities while working on the change task, from the search to the navigation and the editing of code. This will allow a developer to resume more easily from interruptions and speed up the navigation and search in the first place.

Table 2.3: Observations from studies and inferred design implications.

Empirical Observation	**Design Requirement**	#
Developer models are small, abstract and highly connected	Provide adequate abstraction with limited size, focus on highly connected parts and indicate relations	**R1**
Code navigation models are hardly influenced by the size of the actual change, but differ substantially by developer	Provide personalized navigation support	**R2**
Code navigation models exhibit a high structural & lexical cohesion; developers take less time if their navigation is more structured	Combine structural and lexical navigation support and provide proactive structural and lexically context	**R3**
Lexical cohesion of code navigation models is stronger at beginning of exploration	Adapt support and proactive context over time	**R3.1**
Developers use a combination of search and structural navigation	Combine search and navigation	**R4**
Developers frequently revisit code	Indicate and keep track of what was already visited	**R5**
Change task completion time can be several days	Provide a summarization/ abstraction and persistency for previous code context models	**R6**
Change task type influences size of code navigation model	Tailor support to change task type	**R7**

Combined View for Search and Navigation. As found in our studies (**R4**) and by other researchers (e.g., [Sillito et al., 2006,Ko et al., 2006]), developers use a combination of search and navigation when performing a change task. Current tool support in IDEs, however, generally either support search or navigation, requiring developers to switch between views and loose track of dependencies. An initial approach that allows a single query and structural navigation to

expand the search results was presented by Janzen et al. [Janzen and De Volder, 2003]. Since developers usually perform multiple queries for a change task over time and their code context model expands, a combined view should provide support for presenting multiple query and navigation instances at the same time, possibly in a time line view. In addition, to speed up the exploration, search results should be presented with relevant proactive structural context (structural recommendations) and the results should be ranked based on their cohesiveness with respect to the current developer's code context model (**R3**). Initial results on a limited form of structural context for search results already received positive feedback in a study by McMillan et al. [McMillan et al., 2011].

Given the small size and high abstraction of developer's models (**R1**), such a combined view also has to provide an adequate abstraction and summarization. We plan to investigate approaches that aggregate and abstract information over time and yet provide enough and proactive context on the current selection.

Integration of Lexical Dependencies. Along with the split between search and navigation in current IDEs also comes a split between support for following lexical and structural dependencies. While views such as the Package Explorer or the Call Hierarchy in Eclipse allow a developer to follow structural dependencies, the many lexical similarities that exist—and often present a semantic dependency—are neglected in these views. Given the strong lexical cohesion of code context models, tools should more explicitly support this kind of dependency (**R3**) without requiring to switch views and loosing the current working set. Recent research to recommend subsequent navigation steps already leverages the combination of structural and lexical information (e.g [Hill et al., 2007, Lawrance et al., 2008]) to a certain extent. A view of the current code context model should provide explicit cues for these lexical dependencies, indicating the pivotal part of these dependencies, and thus easing the assessment of the relevance of elements and providing a rationale for certain elements in the model. Similarly, when presenting search results or the selected elements, the view should provide proactive lexical context and integrate it with structural context.

Adaptation to Developer and Task. While generally more structural navigation might lead to a better performance, the specific elements and relations that

are being explored for a given change task differ substantially by developer and depend on a lot of factors, such as the developer's experience, and preferences as well as the program comprehension strategy used, such as bottom-up or top-down (e.g. [Brooks, 1978,Brooks, 1983,Pennington, 1987]). As also mentioned by Storey et al. [Storey et al., 1999], tools should support a wide variety of comprehension and navigation activities. So while a tool should provide support for the patterns that exist across developers such as frequent revisitations (**R5**) and the advantage of structural navigation (**R3**), it should also take into account the individual preferences of developers (**R2**). By learning from a developer's past and possibly an interactive component to adjust one's preferences, a tool might be able to provide better recommendations and also a more accurate representation of a developer's code context model of the past.

Since code navigation also varies with the kind of change task as well as over time, an approach should adapt the provided navigation or, more generally, context recommendations based on these factors (**R3.1, R7**). For instance, when a developer starts working on an enhancement and performs a search, the view could provide more structural and lexical context for each result than for a trivial change task. Later on in the task when the developer performs another search, the lexical context would be reduced adapting to the changing behavior as seen in ***O5***. Also, given that certain change tasks are performed over a series of days, for example, with communication happening in between developers to clarify parts of it, there needs to be an option to persist code context models, similarly to the way that Mylyn stores task context, and summaries should be available to ease resuming the task or communicating about it with other developers (**R6**). We plan to conduct studies to further investigate the impact the task type has on the code context models and how to best summarize code context models to resume or share them.

2.6 Related Work

Related work can be categorized into two areas: empirical studies on software developers performing change tasks and tool support for explicit task context.

Empirical Studies. Researchers have extensively observed and studied the program investigation behavior of software developers during maintenance tasks. Ko *et al.* [Ko et al., 2005, Ko et al., 2006] conducted an exploratory study to determine patterns of navigation. They report on 10 developers working on simplistic tasks in a very small system, where they found patterns such as developers starting with a search and then navigating to related elements, collecting small fragments of task-relevant code. Robillard *et al.* [Robillard et al., 2004] conducted an exploratory study to look at the differences in the program investigation behavior of successful and unsuccessful developers. From observing five developers performing a maintenance task on a reasonably-sized system, they found that successful developers reinvestigate methods less frequently and mostly performed structurally guided searches. LaToza *et al.* [LaToza et al., 2007] observed 13 developers working on two tasks on a bigger system, to study how experience affects the program comprehension. They found that experienced developers visit less methods, thus wasting less time on understanding irrelevant methods. Sillito *et al.* [Sillito et al., 2006, Sillito et al., 2005] conducted a laboratory and an industrial study with 25 developers in total working either on a given or on one of their own change tasks. They observed that developers first search for an initial focus point and then explore relationships from these points, also revisiting elements. Based on their observations they identified four types of questions developers ask during change tasks. Starke *et al.* [Starke et al., 2009] focused on the initial investigation phase of a change task and had ten developers perform tasks for 30 minutes. Their observations mainly focused on how participants form search queries and then skim through the results. Wang *et al.* [Wang et al., 2011] conducted an exploratory study with 38 students performing feature location tasks. They focused their study on the initial feature location process and identified search patterns, such as execution-based and exploration-based search. While our results support some of the observations made in the earlier studies, our two studies focus on the actual context models that developers built implicitly for a variety of different tasks and systems and how they overlap with the actual changes they perform for these tasks. In addition, different to most studies mentioned above, we conducted a combination

of an exploratory study with real change tasks on three open source systems and an empirical analysis of data collected from open and closed source developers to validate our findings.

Other studies have observed developers to investigate the process and characteristics of program comprehension. Mayrhauser and Vans [von Mayrhauser and Vans, 1994] used protocol analysis to explore the program comprehension of professional developers working on industrial maintenance tasks. Based on the results of their study, they formulated an integrated model, combining top-down and bottom-up strategies found by other researchers (*e.g.*,, [Brooks, 1978, Brooks, 1983, Pennington, 1987]), to describe the cognitive processes of program comprehension. Corritore and Wiedenbeck [Corritore and Wiedenbeck, 1999] looked at the differences of the mental representation of expert procedural and object-oriented programmers carrying out maintenance tasks on very small systems. Their results show that expert programmers build a mixed mental representation of a program that includes detailed program knowledge as well as domain-based knowledge. Piorkowski et al. [Piorkowski et al., 2012, Lawrance et al., 2008] build upon the theory of information foraging, exploring how developers use *information scent* emitted from *cues* to guide program exploration, and especially study how quickly developers' goals evolve. These approaches focus on the cognitive process of program comprehension, while we investigate the actual code context models, the implicit knowledge, that developers build and retain during comprehension and changing the code.

Explicit Task Context Support. Several approaches provide support for explicitly keeping track of task context—the set of files or code elements a developer works with during a maintenance task. An early tool, Concern Graphs supports developers in recording task context in the form of concern graphs, but requires the developer to manually identify and add relevant elements [Robillard and Murphy, 2002]. Code Bubbles alters the usual IDE editor interface so that each code element a developer navigates to for a task is represented by its own bubble and relations that a developer followed between these bubbles are made explicit [Bragdon et al., 2010]. This way, the context for a task is automatically created when stepping through or editing code. Mylyn, a task-focused UI

approach, differs to these approaches in that it automatically creates an explicit task context from a user's interaction with the development environment [Kersten and Murphy, 2006]. Similarly, DeLine *et al.* proposed to use a user's interaction history for a task to recommend where to navigate next in the code [DeLine et al., 2005b]. All of these approaches specialize in saving task context for elements *after* they have been discovered. We investigate the creation of task context and the design requirements for explicitly supporting developers in the creation and representation of their mental models.

2.7 Conclusion

Software developers currently spend much of their time on change tasks, partially due to the large cost of creating a code context model for each new task assigned to them. In this paper, we have presented the results of two studies, an exploratory study with 12 developers performing change tasks and an empirical analysis of data sets capturing work of professional developers on open and closed source projects. In these two studies we found, amongst other results, that the code navigation models of developers exhibit a high structural and lexical cohesion and that they differ by developer and task. We inferred design implications and presented design consideration for a tool to support the creation and explicit capturing of developer's code context models. In particular, we discuss the combination of search and navigation, the integration of lexical dependencies into the views that currently are predominantly focused on structural dependencies, and the adaptation of such tools to the developer's preferences and the current task. In future work, we plan to develop such an approach, paying particular attention to the presentation and summarization of code context models as well as providing proactive context to speed up the developer in performing change tasks.

3

Towards Activity-Aware Tool Support for Change Tasks

Katja Kevic and Thomas Fritz

Accepted for publication at the 33rd IEEE International Conference on Software Maintenance and Evolution, 2017

Contribution: Study design, data collection, data analysis, and paper writing

Abstract

To complete a change task, software developers perform a number of activities, such as locating and editing the relevant code. While there is a variety of approaches to support developers for change tasks, these approaches mainly focus on a single activity each. Given the wide variety of activities during a

change task, a developer has to keep track and switch a lot between the different approaches. By knowing more about the activities a developer breaks her change task into and when she is working on which activity, we would be able to provide better and more tailored tool support, thereby reducing developer effort.

In our research we investigate the characteristics of these activities, whether they can be identified, and whether we can use this additional information to improve developer support for change tasks. We conducted two exploratory studies with a total of 21 software developers collecting data on activities in the lab and field. An empirical analysis of the data showed, amongst other results, that activities comprise a consistently small amount of code elements across all developers and tasks (approx. 8.7 elements). Further analysis of the data showed, that we can automatically detect the boundaries and types of activities, and that the information on activity types can be used to improve the identification of relevant code elements.

3.1 Introduction

Software developers spend a substantial amount of their time working on change tasks[1], such as bug fixes or enhancements [Perry et al., 1994]. During these change tasks, developers perform a variety of steps and activities as well as address multiple questions as previous research studies have shown [Ko et al., 2006, LaToza et al., 2006, Sillito et al., 2006]. For instance, to complete a bug fixing task, a developer needs to perform a number of distinct activities. First she might start out by locating the bug in the code base using search and navigation. Second, she might thoroughly examine the involved code elements and their relations as well as investigate related documentation to come up with a way to fix the bug, before she finally edits the code and commits the changes to the repository. Each of these activities requires different kinds and granularity of information, ranging from a list of code search results, to individual method

[1] We use the terms *change task* and *task* to refer to any change to a software system that serves a predefined purpose, such as a bug fix or an enhancement.

calls, all the way to API documentation or stackoverflow posts for making code changes.

To support developers in their work on change tasks, various approaches have been proposed to help identify relevant information, such as code search tools for the initial search, recommenders for the code navigation or code completion tools for the editing of code. Predominantly, each of these approaches focuses on supporting one specific activity during the work on a change task to determine relevant information without adapting to the various activities the developer is performing over the course of a change task. With the wide variety of developer activities during a change task [Ko et al., 2006, Sillito et al., 2006] and change tasks lasting anywhere from a few minutes to several days [Fritz et al., 2014b], finding the right approach and the relevant information as well as switching to it is difficult at best [Ko et al., 2007, Minelli et al., 2015].

The more we know about the activities developers perform during a change task, the better we can support them in their work. For instance, if we know that a developer is looking for an initial focus point rather than understanding the specific behavior of an individual code element or trying to edit a code element, we might be able to recommend broader code search results, rather than providing more fine-grained information on code dependencies or code snippets from stackoverflow for the correct editing.

To better support developers in their work on a change tasks and help identify relevant information at the right time, we investigate three questions:

RQ1: What are the characteristics and types of developers' activities on change tasks? Previous research has inferred activities developers perform on various levels of granularity [Ko et al., 2006, Amann et al., 2016, Minelli et al., 2015, LaToza et al., 2006] as well as the questions that developers ask during change tasks [Sillito et al., 2006]. These inferred activities or questions range from low-level developer actions, such as using a navigation tool to high-level questions, such as the dependency between modules. The goal of RQ1 is to extend previous studies by exploring the *set of basic activities that developers' themselves break their work on a change task into* as well as the characteristics of these small units of work, in particular their size, granularity and possible types.

The better we understand in which units developers work and think—what we will refer to as "activities" in the following—the better we can support them in their work and make recommendations on relevant information or tools.

RQ2: How accurately can we automatically detect (a) the boundaries of developers' activities and (b) the types of these activities during a change task? By knowing more about the activity a developer is working on rather than just the high-level change task, we might be able to tailor recommendations to the specific activity and thereby provide more relevant recommendations while also requiring less effort from the developer. The goal of RQ2 is to explore how accurately we can detect the activity a developer is working on and when the developer is switching to another activity during the work on a change task.

RQ3: Can we use activity information to more accurately identify relevant code elements for a change task? The goal of RQ3 is to explore the value of knowing about the activities a developer is working on during a change task. For this, we are focusing on the identification of relevant code elements—a scenario that is commonly addressed in research to support developers during change tasks [Kersten and Murphy, 2006, Coblenz et al., 2006, Robillard and Weigand-Warr, 2005]—and examine whether we can enhance previous approaches by taking advantage of the additional knowledge on the activities that developers are working on.

To address these questions, we conducted two exploratory studies: a field study with nine professional developers working on their own change tasks, and a lab study with twelve developers working on two open source change tasks. For both studies, we collected developers' self-reports on the activities they broke their change tasks into. In addition, we collected their low-level code interactions within their Integrated Development Environment (IDE), such as selections and edits of methods and classes. Based on an empirical analysis of the collected data, we found that there is a small set of basic activity types across all participants that is similar to the ones identified in previous research [Sillito et al., 2006, Vans et al., 1999], including *understanding a specific code element* and *understanding*

a larger context. Also, the self-reported activities exhibit similar characteristics across all types and participants with an activity encompassing approximately 9 code elements (classes and methods) that were explored per activity and a bit more than a third of these being relevant for the activity (RQ1). By applying regression analyses, we found that it is possible to automatically detect different activity types and boundaries—when a developer starts/ends to work on a self-reported activity—with high accuracy (RQ2). Finally, a comparative analysis showed that the use of activity information can improve the precision and recall for recommending relevant code elements by 33% and 57% respectively (RQ3).

This paper makes the following contributions:

- it examines the types and characteristics of developers' self-reported activities for change tasks based on two exploratory studies with 21 developers and related work;
- it demonstrates that an activity's boundaries and type can automatically be detected with high accuracy;
- it illustrates the potential of using activity information to better support developers in their work on change tasks.

3.2 Related Work

Work related to our research can be grouped into empirical studies on developers' activities during change tasks, approaches to identify tasks, and approaches to identify the relevant source code elements for a change task.

3.2.1 Studies on Developers' Activities During Change Tasks

Several empirical studies have been conducted to better understand developers' work on a change task. These studies vary in the granularity of the developer activities they focus on, ranging from very abstract and high-level activities of developers, such as understanding and editing code [Minelli et al., 2015, Ko and Myers, 2005, Vans et al., 1999, Amann et al., 2016, Ko et al., 2006, LaToza et al.,

2006, Singer et al., 1997, Brooks, 1999], and the high-level questions they ask ("To move this feature into this code what else needs to be moved?“ [Sillito et al., 2006]), all the way to very low-level activities, such as microtasks [LaToza et al., 2014] or interactions with code elements (classes, methods and even lines) and commands used [Murphy et al., 2006, Negara et al., 2012, Negara et al., 2013, Amann et al., 2016, Minelli et al., 2015, Beller et al., 2015, Kevic et al., 2015]. Based on an extensive literature survey of empirical studies in the area, we came up with a coding of developer activities that is illustrated in Figure 3.1. Note that this figure presents one way of classifying developer activities that emerged from our coding of previous study findings. It does not capture all activities identified in these studies, also since different studies often used the same terminology for various activities, e.g. navigating and searching [Minelli et al., 2015, Amann et al., 2016] or different terminology for the same activity, e.g. "What is the mapping between these UI types and these model types?“ [Sillito et al., 2006] and gain high level overview of program [Vans et al., 1999].

These empirical studies on developers' activities can also be categorized by the methods used in the studies ranging from fine-grained logging to observations and surveys. Several studies instrumented the IDE to capture and log developers' low-level interactions with code elements and then used these logs to infer developers' activities and time spend on them [Amann et al., 2016, Minelli et al., 2015, Beller et al., 2015]. To capture more of the developers' thought process, several studies used a think-aloud protocol in combination with observations and in-person shadowing or audio recordings to identify the questions developers' ask, and their information needs [Sillito et al., 2006, Ko et al., 2006, Singer et al., 1997]. Finally, researchers have also used surveys to elicit developers' work practices [Singer et al., 1997, LaToza et al., 2006].

In our research, we use a combination of developers' self-reports and low-level interaction logging to explore the activities that developers themselves break their tasks into rather then inferring them retrospectively. Further, we examine the automatic detection and use of these activities.

3.2.2 Task Detection

Researchers have also investigated the manual [Safer and Murphy, 2007] and automatic [Stumpf et al., 2005] detection of whole change tasks. Closest to our work is the approach by Coman and Sillitti [Coman and Sillitti, 2008] that looked at detecting tasks within a recorded navigation sequence. While they evaluated their proposed algorithm in a laboratory setting, Zou and Godfrey [Zou and Godfrey, 2012] reproduced their work in an industrial setting and found that there are levels of activities below a change task for which the detection might be more accurate. Instead of analyzing developer interactions, Barnett et al. [Barnett et al., 2015] examined source code element definitions and usages within change sets to predict when a change set includes unrelated code changes and does not belong to the task.

In our work, we focus on a lower level, namely the activities into which developers break their change tasks into and on how accurately we can detect these automatically.

3.2.3 Relevancy Assessment of Code Elements

Developers spend a substantial amount of their time searching, navigating, and reading source code to locate and keep track of the code elements that are relevant to their work [Ko et al., 2006, Minelli et al., 2015, Amann et al., 2016]. Several approaches emerged to support developers in identifying these relevant code elements more efficiently, for instance, during task resumption [Kersten and Murphy, 2006] or code navigation [Robillard, 2005]. What all of these approaches have in common is a relevancy model that captures the relevancy of individual code elements, yet, the way the relevancy is calculated differs between approaches. Researchers have proposed to use various data sources for creating the relevancy models, ranging from program structure [Biggerstaff et al., 1994, LaToza and Myers, 2011, Augustine et al., 2015], lexical similarities of code elements and change task descriptions [Lawrance et al., 2007], frequency and recency or 'momentum' of developers' code interactions [Kersten and Murphy, 2006, Parnin and Gorg, 2006], version histories [Ying et al., 2004, Zimmermann

et al., 2004], or combinations of some of these sources [Hill et al., 2007]. In a study comparing multiple of these sources, Piorkowski et al. [Piorkowski et al., 2011] found that approaches recommending recently and frequently visited elements performed best.

Different to previous work, we investigate the use of a new kind of information—the information on developer activities—and how we can use this to complement more traditional sources in the identification of relevant code elements.

3.3 Study Method

To investigate how developers decompose change tasks into activities and the automatic detection of these, we conducted a lab and a field study with a total of 21 software developers. The lab study allowed to control for the change tasks and to examine how different developers decompose the same change tasks. In the field study, we examined how professional developers decompose their regular change tasks in a real work environment. In both studies we logged participants' source code interactions in the IDE and gathered data on their activities either through self-reporting or periodic interruptions.

3.3.1 Lab Study

Change Tasks. All participants in this study worked on the same two real change tasks of the open source system Gson [Gson, 2016]. We selected the two change tasks—the feature request with id #42 and the defect with id #153—based on them being already resolved, yet from an actively maintained project, having a commit history as well as their reasonable scope of the solution and the effort required to reproduce/test the task. The change set for feature request #42 included changes to several classes, while the changes to fix defect #153 were located in multiple methods of a single class (see Table 3.1). Gson is a Java project that (re)converts Java objects into JSON strings, has a total of 31.7K lines of Java code not including comments and is composed of 159 Java files.

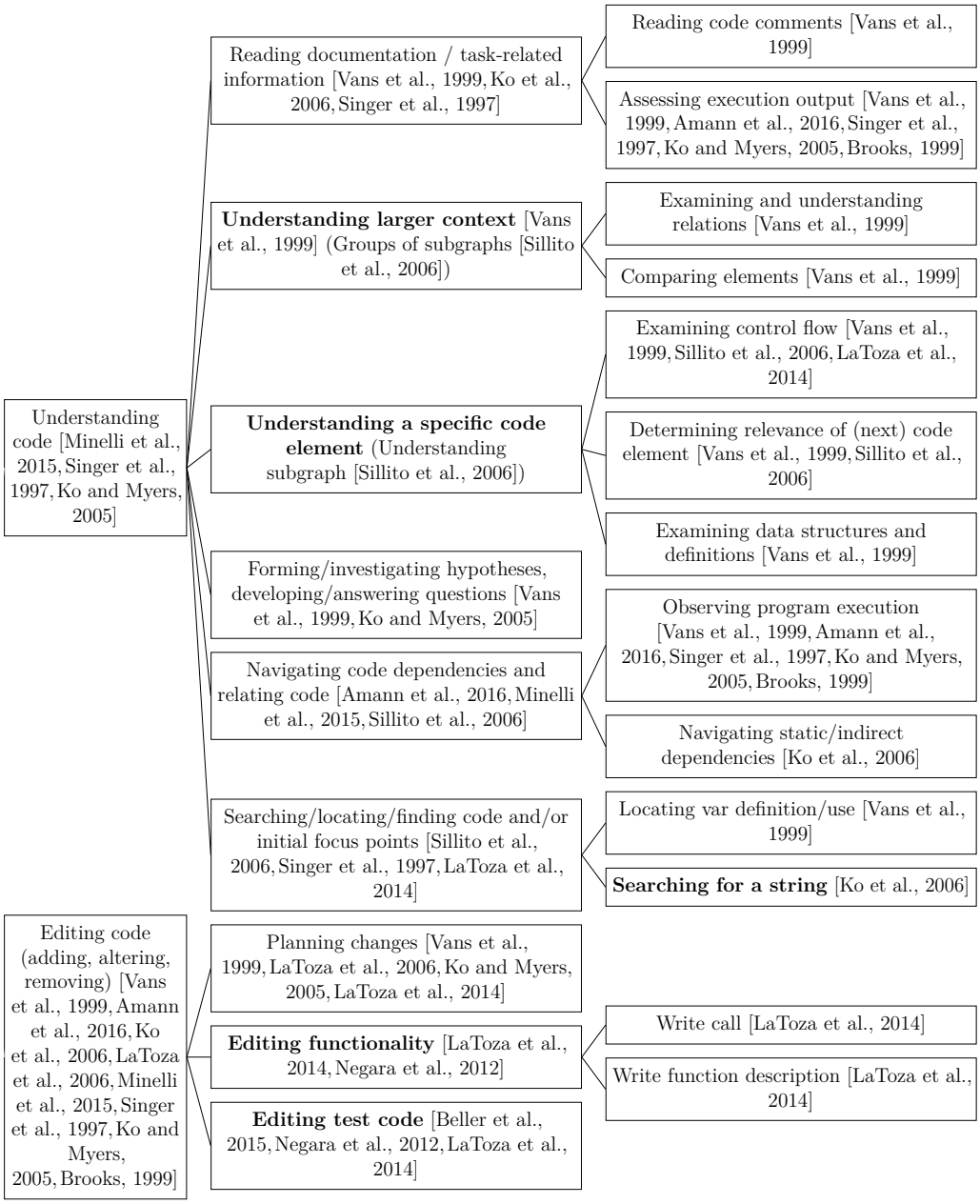


Figure 3.1: Overview of developer activities within the IDE based on a coding of related work. (Elements in bold are the activity types we identified, see section 3.4)

Table 3.1: Tasks used in the lab study.

ID	Date Submitted	Title	Scope of solution committed to the repository
42	9/8/2008	provide a feature to protect against remote "script src" inclusion of Gson output	multiple classes in which multiple methods were affected: `Gson`, `GsonBuilder`, `JsonParserImpl`, `JsonParserImplConstants`, `JsonParserImplTokenManager`
153	9/2/2009	setPrettyPrinting cause missing comma deliminator after an empty map	13 methods in a single class: `JsonPrintFormatter`

Participants. Through personal contacts, we recruited twelve participants (one female, eleven male) from our institution: one postdoctoral researcher, nine graduate and two undergraduate students. All twelve participants had their major in computer science, on average 8.1 (±5.1) years programming and 3.9 (±3.9) years professional programming experience, and were familiar with the Eclipse IDE (eight also stated that Eclipse is the IDE they are most familiar with).

Procedure. The study lasted on average 90 minutes and had a preparation, a training, and a programming phase. In the training phase participants worked on a change task to get familiar with the Gson project. In the programming phase participants worked on a different change task and we gathered detailed data on their activities. We had two groups that we randomly assigned participants to, one group of six started with task #42 (training) and then worked on task #153 (programming), the other group worked on the tasks in reverse order. We changed the order of the tasks to counteract any specific learning effect and to capture activities related to different kinds of change tasks. We prepared an Eclipse IDE instance in which we imported the Gson project into. We also installed a plugin that we developed and that logged all user interactions with code elements within the IDE. This plugin had additional features: a user interface to report

activities during a change task (Figure 3.2), and a user interface to select the code elements that are relevant for an activity.

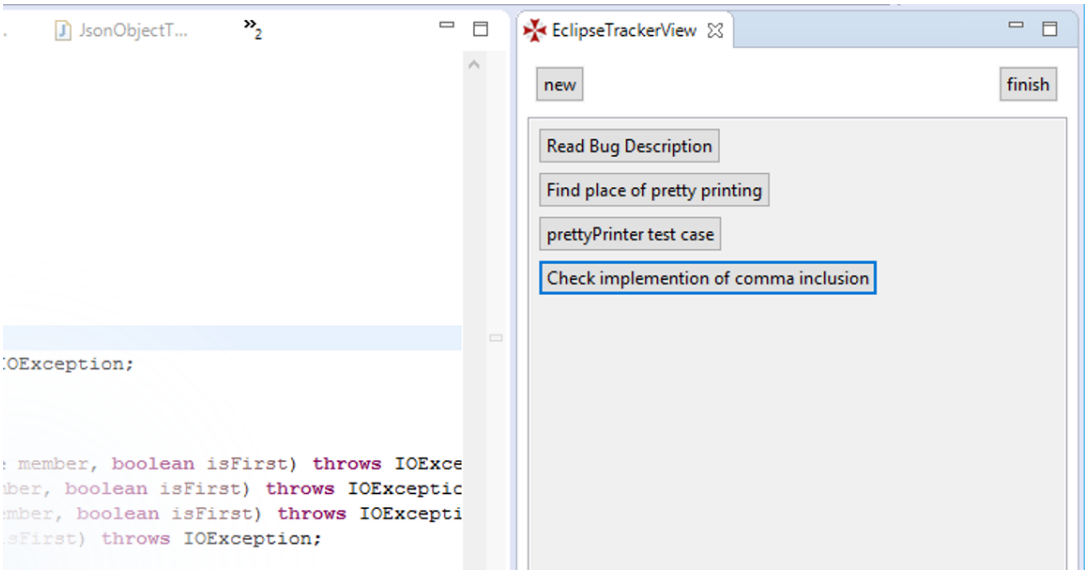


Figure 3.2: Screenshot of our plugin used in the programming phase of the lab study to report and visualize activities.

In the preparation phase, we asked participants to read and sign a consent form and to complete a questionnaire on demographics. We explained that our goal is to study developers' work breakdown for a change task. Then we explained the study procedure, answered questions regarding the procedure, provided a Gson overview, and allowed participants to explore the subject system for a few minutes. In the training phase, we then asked participants to work on the first task for 20 minutes to get more familiar with the code.

In the programming phase, we asked participants to work on the second change task for 25 minutes. For this phase, we asked participants to think-aloud and to report in our plugin "whenever they started working on something new". We intentionally did not specify the phrase "work on something new" any further since we were interested in capturing the units that developers work and think in. In case participants returned to work on something they had previously reported,

they could just select the previously reported activity in the plugin to ease the reporting. We added the think-aloud protocol after we noticed in our pilot study that participants forgot to report some activities without it. After participants worked on the change task for 25 minutes, we asked them to go over all reported activities. Using our plugin, we then presented participants a time-ordered list of the source code elements they interacted with for each activity, and asked them to identify the ones that were relevant for each activity.

3.3.2 Field Study

Change Tasks. All participants in this study worked on their usual change tasks at work. Participants worked on a variety of tasks, such as fixing bugs, adding new features, and writing unit tests. The participants reported that they spend on average 10.5 (±3.4) hours to complete a change task. The projects participants worked on comprised on average 244K (±572K) lines of Java code. For privacy reasons, we cannot disclose any further information on the tasks or projects.

Participants. We used recruiting emails and ended up with nine professional software developers (one female, eight male) from two different medium-sized software development companies and a total of four different development sites where they were either working in an open-plan or in private offices. Participants had an average of 8.6 (±6.0) years of professional programming experience and all nine were using Eclipse for their daily work.

Procedure. The field study had a preparation phase and two working sessions of two hours each, capturing a total of four hours of the work of participants. The working sessions were spread over two different days. In the preparation phase, we again asked participants to read and sign a consent form and to complete a questionnaire on demographics. We further explained that we wish to study how developers decompose change tasks and how developer tools might profit from this knowledge. We then explained the study procedure and gave them the opportunity to ask any questions they had. In the preparation phase, we also installed or asked participants to install our plugin into their IDE. Similar to

the lab study, our plugin logged interactions with source code elements. It also provided features to visualize an ordered list of the code elements they interacted with for the identification of a starting point of the current activity, and allowed to mark the code elements that were relevant for the current activity.

For the two working sessions of two hours each, we picked times when participants had no meetings scheduled, but otherwise asked participants to work as per usual. Most of the participants got at least once interrupted during the study sessions by a colleague, an email which caught their attention or an instant message which they quickly answered. After 30 minutes, we interrupted participants and asked them to:

(a) write down *what they were just working on*,

(b) identify the source code interaction when they started working on the just reported activity using a time-ordered list of their interactions, and

(c) identify the relevant code elements for this activity from the list of source code interactions.

As in the lab study, we did not provide any more detailed specification or examples of what to write down, since we were interested to study and capture how developers themselves decompose the work for a change task. We repeated this procedure a second time in each session and ended up with a total of four interruptions over the two sessions. To minimize the time and impact of our study on their usual work and avoid disrupting them or their colleagues any further, we chose not to employ a think-aloud protocol in the field study.

3.3.3 Data Collection

In both studies, we used our plugin to collect all source code classes and methods that participants selected or edited with in their IDE together with a timestamp for each interaction.

In the lab study, we recorded all reported activity descriptions, the time participants switched to each of these activities, the code elements that participants

interacted with for each activity, and the code elements they selected as being relevant for each activity.

For the field study, we recorded the activity descriptions that participants reported, the starting point, i.e. the first interaction that participants identified and the code elements they selected as being relevant for each reported activity. In total, we gathered data on 37 activities from the nine professional developers, as one developer unsolicitedly extended a working session to capture a further activity. The professional developers in the field study, worked on average 13.06 ($\pm$9.37) minutes on an activity before they got interrupted by the experimenter. The student developers, who reported the activities as they were investigating the change tasks, started a new activity on average each 4.84 ($\pm$3.77) minutes.

3.4 Activity Characteristics (RQ1)

A first step towards activity-aware tool support is to better understand the characteristics and types of activities that developers break their work on a change task into.

Data Analysis. To investigate the characteristics and types of activities, we analyzed the self-reports from our participants. Overall, we collected a total of 96 activities: 37 from the field and 59 from the lab study. For these activities, the authors of this paper first applied an open coding approach to group the reported activities into distinct types of activities. A cross-validation of the activity types by another researcher not involved in this project resulted in an agreement of 92.7% of the activities with the remaining 7.3% being spread across categories. In a second step, we analyzed the number and relevance of code elements that participants interacted with for each self-reported activity. In the following, we report the averages across developers accompanied by the standard deviation denoted as '$\pm$'.

Activity Types. Based on the open coding of the 96 collected activities, we identified six distinct activity types, ranging from the search for a specific string to the changing of test case code. Table 3.2 presents details on each of these six

activity types. For 4 of the 96 collected activity descriptions, we were not able to categorize them due to their vague descriptions.

Table 3.2: The six activity types identified in our two studies together with the number of reported instances of each activity type (# inst), the number of different developers that reported them (#devs), and exemplary instances.

Activity type	#inst. field | lab	#devs field | lab	Exemplary instances
Changes to source code			
#1 Change functionality	8 | 5	6 | 4	(P11): implementation of the permission value connection (S7): Add config flag for global prefix
#2 Change test case code	6 | 10	4 | 7	(P8): Test the upload of user data (S6): Write new test to test the erroneous behavior [..]
Understanding source code			
#3 Underst. a specific code el.	13 | 17	6 | 9	(P2): Check how to read the property pcy.[..].writer (S4): Try to understand Gson.create() method
#4 Underst. a larger context	6 | 17	4 | 9	(P9): [..] why is the data not read correctly? (S4): Find out how I can generate the output that is given [..]
#5 Change task examination	0 | 4	0 | 4	(S10): Inspect task
#6 Searching for specific string	0 | 6	0 | 4	(S11): Search for the setPrettyPrinting
Uncategorized	4 | 0	3 | 0	
	37 | 59		

The first two types we identified address *changes to source code* and differ in the code that was being changed, as the participants in our study explicitly mentioned if they were working on test code. For *changing functionality*, activity descriptions ranged from extending to creating and adding functionality to the code, while *changes to test code* explicitly referred to test code being changed and the functionality that was being tested.

Two activity types refer to the *understanding of source code* and differ in the scope of investigation. While activity instances categorized as *understanding a specific code element* refer to developers trying to understand specific classes or methods, instances of *understanding a larger context* refer to whole features that spread across multiple code elements or large parts of a system that were investigated to locate the root of an undesired behavior or to understand the cause of a change. While the main focus of these two types is on the understanding of code, developers did not just navigate, search, and debug the code, but in several instances also edited the code during their work on these units.

The remaining two types only occurred in the lab study. One type refers to the *examination of a change task* that captured the reading/understanding of the task description and in forming the initial strategy to tackle the change task. The other type focuses on the *search for a specific String*, in particular, the use of a text/code search tool to locate a specific place in the source code. Several field study participants also searched the source code during their work, but different to lab study participants, they did not explicitly differentiate these searches as separate activities. This difference might be due to developers' familiarity with the source code in the field study and the shorter time spend on searching the code.

Overall, our findings provide evidence on a set of reoccurring activity types into which multiple developers decomposed their change tasks into, whether in the lab on open source tasks or in the field on their usual change tasks. The activity types we identified in our studies also overlap with several developer activities identified in previous research as illustrated in Figure 3.1 (bold ones are the ones that overlap), providing further evidence for the generalizability of the identified activity types. At the same time, several activities identified in earlier studies

were not captured by our identified activity types, which stems from two reasons. First, the level of the activities that participants chose in their self-reports is at a higher level than some of the lower-level developer activities identified in previous studies. For instance, while participants navigated the source code using a variety of tools, shortcuts and views, none of them self-reported an activity which focused solely on the navigation activity, but rather used navigation steps in their activities. Second, due to the exploratory nature and the limited number of change tasks captured in our studies, we are not covering the complete range of activities that developers might decompose change tasks into. For instance, none of our participants chose to read external documentation or code comments as identified by previous researchers [Singer et al., 1997, Ko et al., 2006, Vans et al., 1999]. Further studies are needed to extend our set of activity types and characteristics.

Activity Size. The size of self-reported activities is consistently small, with 3.5 (±3.0) unique classes and 5.2 (±4.1) methods that a developer interacted with per activity. This small number of selected and edited source code elements is consistent across all participants and reported activities, with only minimal differences between the professional developers working on their usual tasks in the field study (3.5 ±3.4 classes, 5.0 ±3.6 methods) and the students working on given change tasks (3.5 ±2.8 classes, 5.4 ±4.4 methods). For a fairer comparison, the calculation of these average values does not include activities of the type *change task examination* and *searching for a specific String*, as activities of these two types either included none or very few interactions within the code editor[2]. The activity size also did not vary significantly with the participants' programming experience (Pearson's $r = 0.3$). Only the time spent working on a unit and only for student developers had a statistically significant and moderate effect on the activity size.

Overall, these results suggest that developers decompose change tasks into relatively similar-sized and small activities, regardless of the vast differences in the change tasks they were working on and the number of explored elements. In

[2]Including the activities of the type *searching a specific String* results in 3.3 (±2.7) unique classes and 5.0 (±4.4) unique methods.

contrast, the number of code elements developers explore for a change task can vary considerably from just a few to over hundred [Fritz et al., 2014b] and the number of methods changed can also vary considerably, ranging from a few to nearly 50 [Barnett et al., 2015].

Relevant Code Elements. Across both studies and all reported activities, participants identified only 38.4% of the explored code elements—methods and classes—as relevant to the activity. While developers navigated on average 27.9 ($\pm$23.0) code elements in the field (including revisits) and 18.0 ($\pm$14.5) in the lab, only 2.3 ($\pm$2.1) of the 8.5 unique code elements they navigated to in the field and 2.4 ($\pm$1.3) of the 8.9 unique code elements in the lab were considered relevant. This number is also independent of the time spent on the activity (no significant Pearson correlation) in both studies. Participants mentioned during the study that the irrelevant code elements were often captured when debugging, scrolling files, or when following traces that turned out to be unimportant.

Professional developers discovered the code elements which they assessed to be relevant on average after 9.79 ($\pm$10.07) navigation steps and student developers on average after 4.73 ($\pm$6.86) navigation steps. There is no correlation between developer's experience and the navigation steps performed until relevant code elements were found, meaning that in our analysis, the experience did not account for finding the relevant places in the source code faster. While professional developers identified significantly more methods than classes to be relevant for an activity ($t(36) = 2.9, p = .006$), student developers identified about an equal amount of methods and classes to be relevant.

Developers split change tasks into small and similarly-sized activities that can be categorized into a small set of recurring activity types. For each activity, a developer explored approximately 8.7 code elements of which a third is relevant.

3.5 Detecting Activity Boundaries and Type (RQ2)

To provide activity-aware tool support, we need to be able to detect when a developer switches between activities for a change task (RQ2a) as well as which type of activity the developer is working on (RQ2b). To investigate these two research questions, we analyzed the characteristics of developers' code interaction behavior. In particular, we examined characteristics related to the frequency and recency of code interactions, the relations between successively visited code elements and the edit history (see Table 3.3). Several of these variables have previously been used in other studies to characterize developers' navigation behavior, as indicated in the table.

3.5.1 Boundary Detection

Data Analysis. To detect when a developer started to work on another activity (i.e. an activity switch), we looked at each source code method a developer interacted with and performed stepwise logistic regression. We used the fact whether or not the method interaction was the start of a new activity as dependent variable and the characteristics of the method and previously visited methods as independent variables. In particular, we calculated the independent variables *str_step*, *lex_step*, $field\_step$, *sameClass_step* (Var 2 to 5 in Table 3.3) for up to four interactions back in time[3]. and the variables t, rec, $freq$ (Var 8-10 in Table 3.3), resulting in 19 ((4 x 4) + 3) independent variables. For this analysis, we filtered very short activities—activities that comprised less than four interactions with source code methods—resulting in 4 of the 37 reported activities from the field and 15 of the 59 reported activities from the lab study being excluded. In particular, all instances of the activity types *searching a specific String* and *change task examination* were excluded due to their shortness. In total, we analyzed 420 method interactions that included 33 reported activity

[3]In the exploratory analysis, we also looked further back in the interaction history, but since going back further than 4 interactions did not change the prediction accuracy significantly, we limited it to 4 steps back.

switches from the field study and 435 method interactions from the lab study that included 44 reported activity switches.

Results. The results of our regression models show that developers' code interactions change when they switch to another activity and that we can use this to automatically detect activity switches based on logged interaction history with high accuracy. For the *field*, the final model of the stepwise logistic regression that we applied recognized 96.0% of the method interactions correctly as an activity switch or not. Furthermore, the model correctly detected 25 of the 33 (75.8%) activity switches. The variables that contributed significantly to the detection of the activity switches were the time t elapsed since the last interaction, and whether developers suddenly selected methods that were not using the same fields anymore $field\_step$ over the last four navigation steps. The final model significantly improved upon the baseline model ($\chi^2(2) = 123.288, p < .001$).

For the *lab*, the stepwise logistic regression resulted in a model that is able to correctly predict whether a navigation step is a switch or not in 81.4% of the cases. Out of the 44 explicitly denoted switches, the model recognized 18 (40.9%). The variables that contribute significantly to the final model were whether there is a call relationship to the previously explored method *str_step*, and whether the developer remained in the same class over the last three navigation steps $sameClass\_step$. Our final prediction model again effectuated a significant decrease in unexplained variance compared to the base model, which only includes the intercept ($\chi^2(2) = 9.650, p = .008$).

To further investigate the distance between predicted and recorded switches, we depicted the frequencies of the distances in Figure 3.3. Our model predicted 73 method interactions as switches, with 81% of the predicted switches being very close (less than 5 interactions away) from the one reported by the participants. Part of this slight discrepancy between predicted and recorded switch might be explained by a switch not being at the exact point when a developer reported it, which is also difficult to detect manually. Also, most of the predicted switches that are far away from the recorded activity switch stem from a single participant. These are marked in orange in Figure 3.3 and further investigation is needed to examine this.

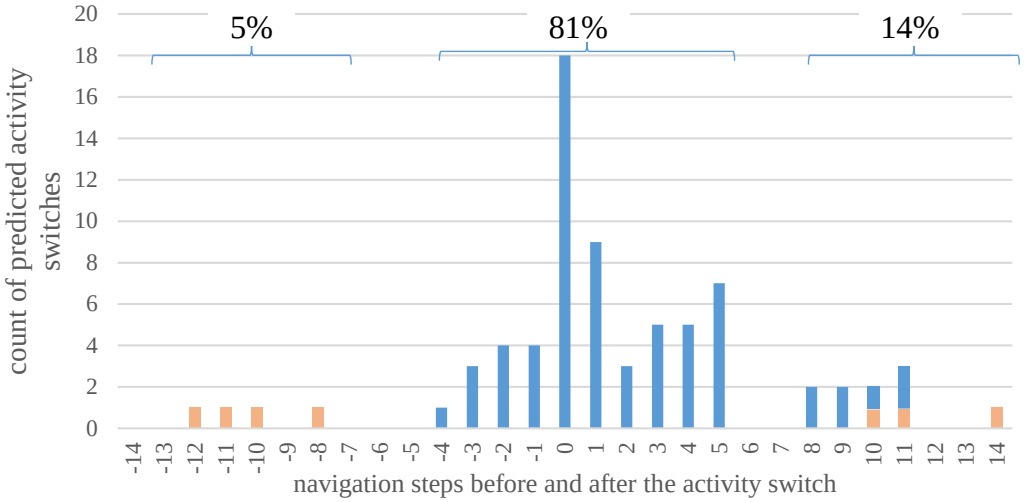


Figure 3.3: Activity switch detection: Distance of predicted to recorded activity switch position, which is at position '0'.

3.5.2 Type Detection

Data Analysis. To analyze the feasibility and accuracy of automatically detecting the types of activities as well as which characteristics are most predictive, we performed a multinomial logistic regression over four of the six activity types. We excluded the types *change task examination* and the *search for a specific String*, since both of these types can be detected without analyzing the code navigation behavior. During *change task examination*, developers had no interactions with the code and were only looking at the task description, which can be detected automatically. During *String searches*, developers had much less code interactions (on average only 1.7 ±1.2 code methods) than during their work on the other four activity types and interacted with a search tool in the IDE, which can also be detected automatically.

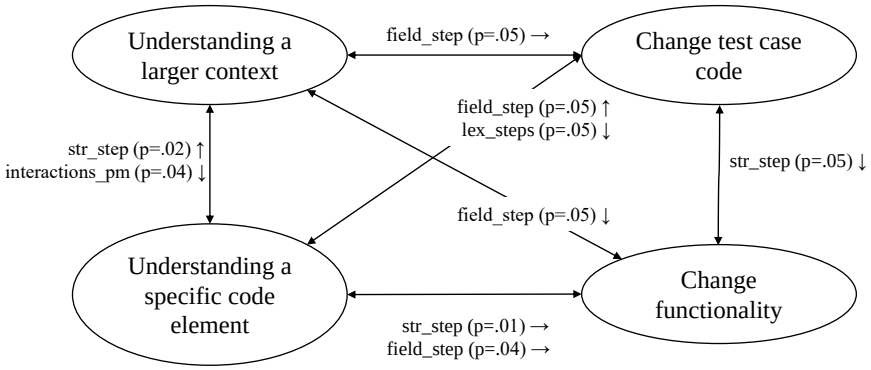
For the regression analysis, we used the characteristics and relations between successively visited methods and the edit history (Var 1-8 in Table 3.3) as independent variables to predict the activity type (dependent variable). We calculated these characteristics for the method navigations a developer performed

Table 3.3: Variables used to describe developers' navigation behavior.

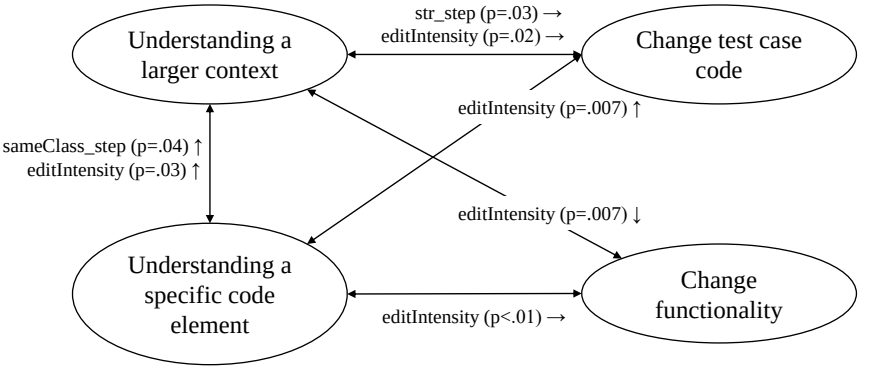
Variable	VarID	Description
interactions_pm	1	The amount of interactions within the source code per minute.
str_step	2	Determines whether a call relationship is existent between two methods, e.g., [Robillard and Murphy, 2003]
lex_step	3	Determines the cosine similarity between two methods, e.g., [Fritz et al., 2014b]
field_step	4	Determines whether two methods use the same field, e.g., [Robillard and Murphy, 2003]
sameClass_step	5	Determines whether two methods are defined within the same class, e.g., [Fritz et al., 2014b]
isEdited	6	Determines whether the method was edited, e.g., [Kersten and Murphy, 2006]
editIntensity	7	Determines the amount of characters which were changed within the method, e.g., [Kersten and Murphy, 2006]
t	8	The time elapsed since the last interaction, e.g., [Coman and Sillitti, 2008]
rec	9	Determines how recent (in terms of navigation steps) a method was selected, e.g., [Kersten and Murphy, 2006]
freq	10	Determines how frequent a method was selected, e.g., [Kersten and Murphy, 2006]

for an activity since she started working on the activity. In particular, we calculated the relative frequencies of the different kinds of navigation relations during the work on an activity, the average cosine similarity between subsequently selected methods, if a method was edited or not, and the amount of characters changed overall.

Results. The results of our analysis show that we can automatically detect different activity types with high accuracy by again using characteristics of developers' code interaction history. For the *field*, the final model of the performed regression analysis achieved a total prediction accuracy of 82%, with only few



(a) Field study.



(b) Lab study.

Figure 3.4: Significant variables for predicting an activity type in the field and lab study. The arrows next to the variable names point to the types in which the variable value is higher.

false predictions of each activity type. Compared to a baseline model, i.e. a model that omits all variables and only uses the intercept, the final regression model achieved a significant decrease in unexplained variance of $\chi^2(12) = 56.21, p < .001$. The step-wise multinomial logistic regression analysis showed that each of the four variables *str_step*, *field_step*, *interactions_pm*, and *lex_step* provides a significant contribution to the final model for predicting the activity type. Figure 3.4a presents the variables that provide a significant contribution to distinguish between pairs of activity types, with the arrows indicating for which type a variable is higher. For example, developers who were working on *understanding a specific code element* interacted with more elements per minute

than when they were *understanding a larger context* in the source code, but followed generally less call dependencies (*str_step*).

For the *lab*, the multinomial logistic regression that we carried out on the gathered activities resulted in a model with 79.6% accuracy for the 49 activities with only few activity types being predicted incorrectly. The final regression model was again better than the baseline model and achieved a significant decrease in unexplained variance of $\chi^2(12) = 68.03, p < .001$. The distinguishing variables in this model were *str_step*, $field\_step$, $sameClass\_step$, and $editIntensity$. Figure 3.4b presents the variables that help significantly to distinguish between two types of activities.

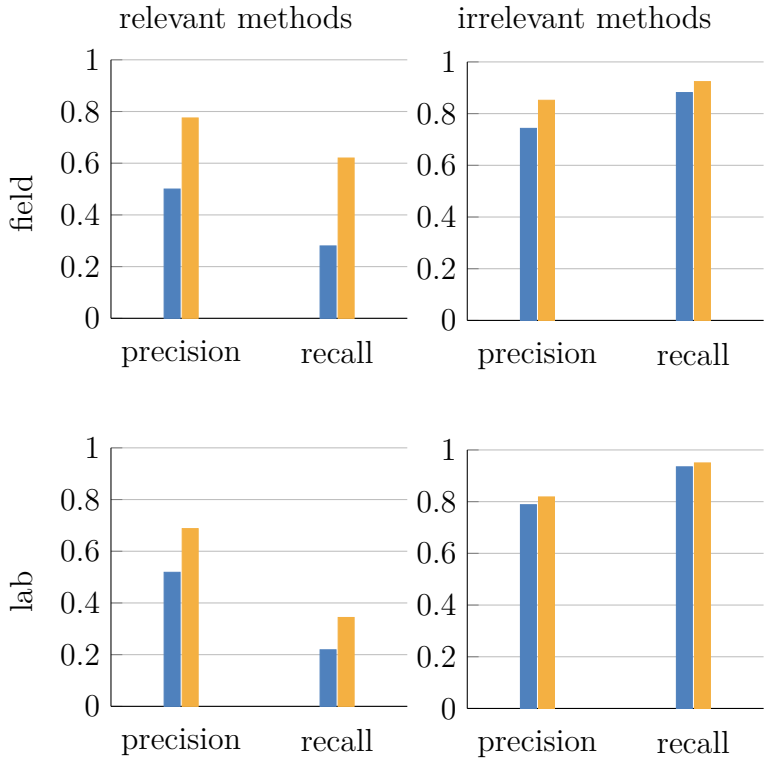
While these models provide a good indication of the feasibility and the accuracy that can be achieved, especially since the accuracy is similar in both studies, and the variables indicate some of the differences in activity types, more data is needed to make these results more generalizable. For instance, the amount of changed characters ($editIntensity$) was a significant variable in the lab study, but not in the field study. We believe that the unfamiliarity of participants with the system in the lab study and the substantially shorter time that participants spent on an activity, might have led to faster code changes and trial and error behavior in the lab that could account for this difference. Another reason for the difference in the variables used for the final model are the uneven distribution across the activity types in the studies. Since only 10.2% of the activities were of type *change functionality* in the lab study and the other 3 types each occurred in more than 20% of the cases, the significance of the variables might have been influenced.

Activity boundaries and activity type can be detected automatically and with high accuracy using fine-grained characteristics of developers' code navigation.

3.6 Activity-Aware Relevancy Models (RQ3)

To explore the value of activity information for providing better tool support to developers, we focus on the identification of relevant code elements—a scenario

Table 3.4: Identification of relevant and irrelevant methods without (■) and with (■) activity information.



commonly addressed by researchers [Kersten and Murphy, 2006, Biegel et al., 2015, Bragdon et al., 2010, Coblenz et al., 2006, Robillard and Weigand-Warr, 2005]—and examine whether activity information improves the precision and recall of automatically identifying relevant code elements.

Data Analysis. To compare the detection of relevant elements with and without knowledge on the activities, we performed logistic regression and trained two types of classifiers: a base case and a set of activity-aware classifiers. For the base case, we used participants' relevance assessments of code elements as the dependent variable and characteristics of developers' code interactions as independent variables. For the independent variables, we focused on characteristics that

previous research suggested for predicting relevant code elements. In particular, we calculated the variables *str_step*, *field_step*, *sameClass_step*, *lex_step*, *isEdited*, *editIntensity*, *rec*, and *freq* (Var 2-7, 9 and 10 of Table 3.3) for each method a participant interacted with.

To simulate activity-aware relevance prediction, we used the same dependent and independent variables as for the base case, but instead of just training one classifier over all code interactions in the training set, we trained one classifier per activity type and only over the the code interactions that were performed during the specific activity type in the training set. For these activity-aware classifiers, we focused on the four activities *change functionality*, *change test case code*, *understanding a specific code element*, and *understanding a larger context*, since activities of the other two types were again too short (having too few code interactions).

Overall, we performed a 10-fold cross validation using a random 90% of the data for training and the remaining 10% of the data for testing. To compare the prediction of relevant code elements, we then predicted the relevance for each source code method in the test set once with the base case classifier and once with the classifier of the respective activity type. By having one classifier per activity type, we simulate an activity-aware classifier that knows which type of activity a developer is currently performing and when she is switching.

Results. The results of our analysis show that activity information considerably improved precision and recall for identifying relevant and irrelevant methods within developers' navigation histories (see Table 3.4). For the *field study*, the precision of identifying relevant source code methods increased by 55.0% and the recall by 121.43%. For the *lab study*, the precision and recall to identify relevant methods increased by 32.59% and 57.14% respectively. The results of our analysis of relevant methods also shows that different variables play a more or less significant role in different activity types. For example, developers are more interested in structurally connected methods when they are *changing functionality* but less so when they are trying to *understand a specific code element*. While our analysis was conducted retrospectively, the insights into the contribution of different variables to the regression models for different activity

types can also be used in the future to improve online recommendations on where to navigate next.

Knowledge of developers' activities can be used to improve tool support for change tasks.

3.7 Threats to Validity

External Validity. There are several threats to the external validity of our results, in particular the limited number of study participants and change tasks in the lab study, the focus on one programming language and IDE and the unfamiliarity of the lab participants with the project. We tried to mitigate these threats by combining the controlled lab study with a field study in which professional developers worked on their own change tasks and their usual projects that differed in size and domain, and by choosing realistic change tasks for the lab study and a study system which is comparatively well commented.

Internal Validity. The presence of the experimenter, the regular interruptions, the writing of activity descriptions and the discomfort of being observed during the study might have influenced the participants' navigation behavior. By restricting the interaction with the participants during the study to a minimum, we tried to minimize the effect of this threat.

Construct Validity. The open coding of the gathered activity descriptions might contain a subjective bias. To mitigate this risk, all authors of this paper independently determined activity types after iterating through the activity descriptions. One of the authors then coded the activity into the emerged categories. To cross-validate our coding, we asked a working colleague to do the same. The two codings overlapped in most cases and in the few cases it did not, we discussed and finally agreed on one that was then used for our analysis.

3.8 Discussion

Our findings confirm and extend the existing body of research on how developers decompose tasks into the smaller activities they think and work in. The findings also demonstrate the potential of automatically detecting these activities and on the beneficial use of activity information. In the following, we will discuss the runtime detection of activities and how activities can further be used to support developers in their work on tasks.

3.8.1 Runtime Detection

The earlier we can determine the type and boundaries of activities, the better we can take advantage of the information and support developers in their work. Our findings already demonstrate that it is possible to automatically detect activity boundaries with high accuracy at runtime by only taking into account developers' past code interactions. For the detection of the activity type on the other hand, we performed an analysis that uses all code interactions for an activity and could thus only be done retrospectively after an activity is completed. In a preliminary analysis that only considered the first five interaction events of the approximately 28 that a developer performs per activity, we found that it is possible to predict the activity type with an accuracy of more than 63% for both, the field and the lab study. While further analysis is needed, these results already suggest that we are able to achieve a high accuracy for the automatic detection and take advantage of it early on.

3.8.2 Better Developer Support

Dynamic Adaptation of Artifact Recommendations. Several approaches have been proposed to support developers during change tasks by recommending various kinds of artifacts, ranging from specific source code elements, all the way to posts on Q&A sites, such as Stackoverflow. While all of these recommendations can be useful at some point during a change task, providing too many of them can limit the usefulness and lead to information overload. Taking into account when

a developer starts to work on a particular activity type, we should be able to better tailor recommendations to the developers' information needs at any point in time and improve the usefulness of the recommendations. For example, while documentation might be more valuable when a developer is trying to understand a larger context, specific code snippets might be more useful when she is changing functionality. Similarly, activity information could be used to dynamically tailor views and the presented code context within an IDE that is often not adequate to support software developers as Minelli et al. [Minelli et al., 2016] pointed out. Our analysis of activity types showed that different kinds of code information are relevant for different activity types. For instance, when a developer is changing functionality, views in the IDE could be organized to highlight the structural relations that are particularly relevant for this activity type.

Interruptions and Task Resumption. Interruptions by coworkers, instant messages or email occur frequently for software developers. When these interruptions happen at an inappropriate time, it takes the developer a comparatively high effort to get back into the train of thought and errors are more likely to happen [Bailey and Konstan, 2006]. Previous research on interruptions has already shown that interruption costs are considerably lower when developers switch between (sub-)tasks than when they are in the middle of a task, rendering these switches as more appropriate moments for interruptions [Iqbal et al., 2004]. Since entire change tasks can last several hours or even days, deferring an interruption until the next task switch might not be feasible. Activity switches are a lot more frequent (approximately every 8.4 minutes in our studies), denote switches that developers perceive in their work, and can be detected automatically with high accuracy. This suggests that activity switches represent good moments for interruptions and might be used to minimize required effort and potential errors during work.

At the same time, interruptions at inopportune moments are not completely avoidable. Information on the activity boundaries, types and the relevant elements within could be used to ease task resumption. In particular, the activity information could be used to provide a better overview of the work to be re-

sumed, for example by highlighting the relevant elements, or by presenting a more high-level activity view of the code interaction.

3.9 Conclusion

In a first step towards activity-aware tool support for change tasks, we investigated the activities that developers break their work on a change task into. We conducted two exploratory studies, a lab and a field study with a total of 21 software developers and examined the characteristics, the automatic detection and the potential value of these activities and the knowledge thereof. Our results show that activities are consistently small across developers and change tasks and that few of the code elements that developers interacted with during an activity are relevant. Our analysis also showed that fine-grained navigation behavior of developers can be used to accurately detect activity boundaries and types.

The newly gained insights on the activities of developers and their automatic detection represent valuable opportunities to better support developers in their work. In particular, this information can be used to improve the detection and recommendation of relevant code elements and artifacts as well as for better interruption management and task resumption. A case study we performed on the detection of relevant code elements has shown that this information can be used to improve upon more traditional approaches by 33% for precision and 57% for recall, indicating the potential that the detection of activities can have on developer support.

4

Eye Gaze and Interaction Contexts for Change Tasks — Observations and Potential

Katja Kevic, Braden M. Walters, Timothy R. Shaffer,
Bonita Sharif, David C. Shepherd and Thomas Fritz
Published in the Journal of Systems and Software, 2017, as extension of the publication at the 10th Joint Meeting on Foundations of Software Engineering, 2015
Contribution: Study design, data collection, data analysis, and paper writing

Abstract

The more we know about software developers' detailed navigation behavior for change tasks, the better we are able to provide effective tool support. Currently, most empirical studies on developers performing change tasks are, however, limited to very small code snippets or limited by the granularity and detail of the data collected on developer's navigation behavior. In our research, we extend this work by combining user interaction monitoring to gather interaction context-the code elements a developer selects and edits -with eye-tracking to gather more detailed and fine-granular gaze context-code elements a developer looked at. In a study with 12 professional and 10 student developers we gathered interaction and gaze contexts from participants working on three change tasks of an open source system. Based on an analysis of the data we found, amongst other results, that gaze context captures different aspects than interaction context and that developers only read small portions of code elements. We further explore the potential of the more detailed and fine-granular data by examining the use of the captured change task context to predict perceived task difficulty and to provide better and more fine-grained navigation recommendations. We discuss our findings and their implications for better tool support.

4.1 Introduction

Developers spend a significant amount of their time searching, navigating and reading source code to find and modify the elements relevant to a change task at hand [Ko et al., 2006]. During this code exploration a developer gradually builds up an implicit change task context that consists of all the explored source code elements and relations. While these task contexts often stay implicit [LaToza et al., 2006], there has been a shift towards automatically capturing task context based on a developer's interactions with the code elements in an integrated development environment (IDE) [Kersten and Murphy, 2006, Robbes and Lanza, 2008, Mylyn, 2015]. The more we know about task contexts and the code a developer explores for a change task, the better we are able to develop effective

tool support for a variety of programming activities, such as proactive navigation recommendation (e.g., [Robillard, 2005, Piorkowski et al., 2012]) or task resumption support (e.g., [Kersten and Murphy, 2006]).

Recent advances in technology, such as eye-tracking devices, afford new opportunities to collect more detailed information on a developer and her work. Studies using eye-tracking sensors and other biometric sensors have generated new insights on, for instance, brain activation patterns [Siegmund et al., 2014], developers' perceptions of difficulty [Fritz et al., 2014a], and the ease of comprehending different representations of code [Sharif and Maletic, 2010a, Bednarik, 2012]. Yet, these studies predominantly focus on small code snippets of the size of source code methods and do not capture contexts of real-world change tasks. Additionally they fail to leverage established methods of collecting interaction data, such as instrumenting the IDE and automatically mapping x,y coordinates back to source code elements, and thus often produce data that is difficult to analyze.

In our research, we extend previous work by taking advantage of eye-tracking technology and examining developers' fine-grained code exploration behavior for realistic change tasks. By using an eye-tracker and capturing a developer's eye gazes on line and statement level, we are able to gather much deeper insight into a developer's code exploration behavior than existing techniques that operate on file or method granularity. This type of information is particularly valuable since developers spend a considerable amount of their time reading individual source code methods [Ko et al., 2006]. In addition to the analysis of a developer's fine-grained navigation behavior, using an eye-tracking approach also enables us to answer questions on the difference in the data captured through eye-tracking and interaction logging and how such fine-grained data can be used to better support developers.

To investigate the fine-grained navigation behavior for realistic change tasks, we conducted a study with 12 professional and 10 student developers. In this study we used interaction monitoring in combination with eye-tracking and simultaneously captured all code elements a developer selected or edited—*interaction context*—and all code elements a developer looked at—*gaze context*—while they

were working on three change tasks from an open source software system. While the interaction context includes source code elements on method-level or higher granularity, the gaze context was captured on statement and line-level.

Based on the analysis of the detailed code exploration traces of developers, we made observations on developers' task contexts from different perspectives. Our analysis on the within method navigation behavior revealed that developers only read few lines of a method and that these lines are generally connected through data flow. Our analysis on the navigation behavior between methods revealed that developers frequently switch to methods in close proximity or within the same class and that they only focus on few methods thoroughly. We further found that developers either use a skimming or a seeking strategy to explore the source code for a change task and that developers who solved a change task successfully read more methods thoroughly. In our analysis we also investigated the differences between these two kinds of contexts and found that the gaze context not only captures more and more fine-grained source code elements, but also different aspects about the developers' navigation.

We further explore the potential application of this new fine-granular data source in two scenarios: line-level navigation recommendation and the prediction of task difficulty. In an empirical analysis of the data captured in our study, we found that out of four models based on data flow, proximity, recency and frequency, the proximity-model works best and allows to predict the next line visited by a developer in 73.6% of the cases. These results can be used to inform, for example, navigation tool support for summarizing methods or highlighting parts therefore in the context of a change task. For the prediction of task difficulty, we conducted a second empirical analysis that focused on predicting the difficulty of the current change task based on specific characteristics of a developer's navigation behavior, such as the number of line switches or the number of switches between methods. We found that gaze context can be used to more accurately predict task difficulty, and that for both, interaction and gaze context, a developer's navigations to methods right above or underneath a current method can be used to predict task difficulty best.

The contributions of this paper are summarized as follows:

- Study findings based on eye-tracking and user interaction monitoring that provide insights into the detailed navigation behavior of 22 developers working on realistic change tasks.

- An approach to automatically and on-the-fly capture the fine-grained source code elements a developer looks at in an IDE while working with large files, thereby significantly improving current state-of-the-art that limits eye tracking studies to only single methods.

- Analysis of different strategies developers use for code exploration with respect to successful and unsuccessful results

- Potential uses of the fine-grained eye tracking data for navigation recommendations and predicting task difficulty.

- A discussion on the value of the data gathered and the opportunities the data and the findings offer for better developer support.

In this paper, we extend our previous work [Kevic et al., 2015] by analyzing and investigating the different strategies developers employ for code exploration during change tasks as well as an analysis of successful versus unsuccessful code exploration (see Section 4.4.4). In addition, we also go a step further and look at potential uses of the fine-grained data in two scenarios, one on line-level navigation recommendation and one on the prediction of task difficulty (see Section 4.5).

The paper is organized as follows. First, we provide an overview of related work (Section 4.2), before we describe the exploratory study and how we collected the data (Section 4.3). In Section 4.4, we present the results of our study in a form of observations, and in Section 4.5 we explore the potential use of the fine-grained data in two different scenarios. Section 4.6 lists threats to validity and Section 4.7 discusses our observations and future ideas before we conclude in Section 4.8.

4.2 Related Work

Our work can be seen as an evolution of techniques to empirically study software developers working on change tasks. Therefore, we classify related work roughly along its evolution into three categories: manual capturing, user interaction monitoring, and biometric sensing of developers' work.

Manual Capturing. Researchers have been conducting empirical studies of software developers for a very long time. Many of the earlier studies focused on capturing answers of participants after performing small tasks to investigate code comprehension and knowledge acquisition (e.g., [Brooks, 1983, Shneiderman and Mayer, 1979, Rist, 1986]). Later on, researchers started to manually capture more detailed data on developers' actions. Altmann, for instance, analyzed a ten minute interval of an expert programmer performing a task and used computational simulation to study the near-term memory [Altmann, 2001]. Perhaps one of the most well-known user studies from this category is the study by Ko et al. [Ko et al., 2006]. In this study, the authors screen captured ten developers' desktops while they worked on five tasks on a toy-sized program and then hand-coded and analyzed each 70 minute session. In a study on developers performing more realistic change tasks, Fritz et al. [Fritz et al., 2014b] used a similar technique and manually transcribed and coded the screen captured videos of all participants. While all of these studies are a valuable source of learning and led to interesting findings, the cost of hand-coding a developers' actions is very high, which led to only a limited number of studies providing detailed insights on a developers' behavior.

User Interaction Monitoring. More recently, approaches have been developed to automatically capture user interaction data within an IDE, such as Mylyn [Mylyn, 2015, Kersten and Murphy, 2005, Kersten and Murphy, 2006]. Based on such automatically captured interaction histories—logs of the code elements a developer interacted with along with a timestamp—researchers have, for instance, investigated how developers work in an IDE [Murphy et al., 2006], how they navigate through code [Parnin and Gorg, 2006, Piorkowski et al., 2011, Augustine et al., 2015], or how developers' micro interaction patterns might

be used for defect prediction [Lee et al., 2011]. Even the Eclipse team themselves undertook a major data collection project called the Usage Data Collector that, at its peak, collected data from thousands of developers using Eclipse. Overall, the automatic monitoring of user interactions was able to significantly reduce the cost for certain empirical studies. However, these studies are limited to the granularity and detail of the monitoring approach. In case of user interaction monitoring, the granularity is predominately the method or class file level and detailed information, such as the time a developer spends reading a code element or when the developer is not looking at the screen, is missing and makes it more difficult to fully understand the developers' traces.

Biometric Sensing. In parallel to the IDE instrumentation efforts, researchers in the software development domain have also started to take advantage of the maturing of biometric sensors. Most of this research focuses on eye-tracking [Rayner, 1998, Just and Carpenter, 1980], while only few studies have been conducted so far that also use other signals, such as an fMRI to identify brain activation patterns for small comprehension tasks [Siegmund et al., 2014], or a combination of eye-tracking, EDA, and EEG sensors to measure aspects such as task difficulty, developers' emotions and progress, or interruptibility [Fritz et al., 2014a, Müller and Fritz, 2015, Züger and Fritz, 2015].

By using eye-tracking and automatically capturing where a developer is looking (eye gaze), researchers were able to gain deeper insights into developers' code comprehension. One of the first eye-tracking studies in program comprehension was conducted by Crosby et al., who found that experts and novices differ in the way they looked at English and Pascal versions of an algorithm [Crosby and Stelovsky, 1990]. Since then, several researchers have used eye-tracking to evaluate the impact of developers' eye gaze on comprehension for different kinds of representations and visualizations such as 3D visualizations [Sharif et al., 2013], UML diagrams [Yusuf et al., 2007, De Smet et al., 2014], design pattern layout [Sharif and Maletic, 2010b], programming languages [Turner et al., 2014], and identifier styles [Sharif and Maletic, 2010a, Binkley et al., 2013]. Researchers have also used eye-tracking to investigate developers' scan patterns for very small code snippets, finding that participants first read the entire code snippet to get

an idea of the program [Uwano et al., 2006]. Other researchers examined different strategies novice and expert developers employ in program comprehension and debugging [Bednarik and Tukiainen, 2006, Bednarik, 2012], as well as where developers spend most time when reading a method to devise a better method summarization technique [Rodeghero et al., 2014]. Finally, researchers have also used eye-tracking to evaluate its potential for detecting software traceability links [Sharif and Kagdi, 2011, Walters et al., 2013, Walters et al., 2014]. All of these studies are limited to very small, toy applications or single page code tasks. Furthermore, in many of these studies, the link between the eye gaze (e.g. a developer looking at pixel 100, 201 on the screen) to the elements in an IDE (e.g., a variable declaration in line 5 of method OpenFile) had to be done manually.

To the best of our knowledge, this paper presents the first study on realistic change task investigation that collects and analyzes both, developers' user interaction and eye gaze data. Due to the approach we developed that automatically links eye gaze data to the underlying source code elements in the IDE, we reduce the need of manual mapping and are able to overcome the single page code task limitation of previous studies, allowing for change tasks on a realistic-sized code base with developers being able to naturally scroll and switch editor windows.

4.3 Exploratory Study

We conducted an exploratory study with 22 participants to investigate the detailed navigation behavior of developers for realistic change tasks. Each participant was asked to work for a total of 60 minutes on three change tasks of the open source system *JabRef* in the Eclipse IDE, while we tracked their eyes and monitored their interaction in the IDE. For the eye-tracking part, we developed a new version of our Eclipse plugin called iTrace [Walters et al., 2014], by adding automatic linking between the eye gazes captured by the eye-tracking system to the underlying fine-grained source code elements in the IDE in real-time. All study materials are available on our website [iTrace, 2015].

4.3.1 Procedure

The study was conducted in two steps at two physical locations. In the first step, we conducted the study with twelve professional developers on site at ABB. We used a silent and interruption free room that was provided to us for this purpose. In the second step, we conducted the study with ten students in a university lab at Youngstown State University. We used the same procedure as outlined below at both locations.

When a participant arrived at the study location, we asked her to read and sign the consent form and fill out the background questionnaire on their previous experience with programming, Java, bug fixing and Eclipse. Then, we provided each participant a document with the study instructions and a short description of JabRef. Participants were encouraged to ask questions at this stage to make sure they understood what they were required to do during the study. The entire procedure of the study was also explained to them by a moderator. In particular, participants were told that they will be given three bug reports from the JabRef repository and the goal was to fix the bug if possible. However, we did mention that the ultimate goal was the process they used to eventually fix the bug and not the final bug fix.

For the study, participants were seated in front of a 24-inch LCD monitor. When they were ready to start, we first performed a calibration for the eye-tracker within iTrace. Before every eye-tracking study, it is necessary to calibrate the system to each participants' eyes in order to properly record gaze data. Once the system was successfully calibrated, the moderator turned on iTrace and Mylyn to start collecting both types of data while the participants worked on the change tasks. Participants were given time to work on a sample task before we started the one hour study on the three main tasks. At the end of each change task, we had a time-stamped eye gaze session of line-level data and the Mylyn task interactions saved in a file for later processing. We also asked each participant to type their answer (the class(es)/method (s)/attribute(s) where they might fix the bug) in a text file in Eclipse at the end of each change task.

For the study, each participant had Eclipse with iTrace and Mylyn plugins installed, the JabRef source code, a command prompt with instructions on how

to build and run JabRef, and sample bib files to test and run JabRef. There were no additional plugins installed in Eclipse. The study was conducted on a Windows machine. Each participant was able to make any necessary edits to the JabRef code and run it. They were also able to switch back and forth between the Eclipse IDE and the JabRef application. iTrace detects when the Eclipse perspective is in focus and only then collects eye gaze data. We asked subjects not to resize the Eclipse window to maintain the same full screen setup for all subjects and not to browse the web for answers since we wanted to control for any other factors that might affect our results.

4.3.2 Participants

For our study, we acquired two sets of participants: twelve professional developers working at ABB Inc. that spend most of their time developing and debugging production software, and ten undergraduate and graduate computer science students from Youngstown State University. Participants were recruited through personal contacts and a recruiting email. All participants were compensated with a gift card for their participation.

All professional developers reported having more than five years of programming experience. Seven of the twelve reported having more than five years of experience programming in Java, while the other five reported having about one year of Java programming experience. Nine of the twelve professional participants also rated their bug fixing skills as above average or excellent. With respect to IDE usage, four of the twelve stated that they mainly use Visual Studio for work purposes and that they were not familiar with the Eclipse IDE, and one participant commented on mainly being a vim and command line user. Of the twelve professional developers, two were female and ten were male.

Among the ten student participants, one participant had more than five years of programming experience, five students had between three and five years programming experience, and four of them had less than two years programming experience. Three of the students reported having between three and five years of Java programming experience, while seven students had less than two years. Three of the ten students rated their bug fixing skills as above average, and seven

rated them as average. All but one student stated that they were familiar with the Eclipse IDE. Of the ten students, one was female and nine male.

4.3.3 Subject System and Change Tasks

We chose *JabRef* as the subject system in this study. JabRef is a graphical application for managing bibliographic databases that uses the standard LaTeX bibliographic format BibTeX, and can also import and export many other formats. JabRef is an open source, Java based system available on SourceForge [JabRef, 2015] and consists of approximately 38 KLOC spread across 311 files. The version of JabRef used in our study was 1.8.1, release date 9/16/2005. We chose an earlier release of JabRef to ensure that there was a sufficient number of resolved change tasks available for us to choose study tasks from and that had change sets associated with them.

To have realistic change tasks in our study, we took the tasks directly from the bug descriptions submitted to JabRef on Sourceforge. Information about each task is provided in Table 4.1. All of these change tasks represent actual JabRef tasks that were reported by someone on Sourceforge and that were eventually fixed in a later JabRef release. The only criteria for selecting tasks was that they had to address a change in the source code of the system and, for instance, not in the configuration files. We randomly selected tasks from a list of closed bug reports that fulfilled this criteria and that also varied in difficulty as determined by the scope of the solution implemented in the repository.

A time limit of 20 minutes was placed for each task so that participants would work on all three tasks during the one hour study. To familiarize participants with the process and the code base, each participant was also given a sample task before starting with the three main tasks for which we did not analyze the tracked data. The task order of the three tasks was randomly chosen for each participant.

Table 4.1: Tasks used in the study.

ID	Bug ID	Date Submitted	Title	Scope of Solution in Repository
T2	1436014	2/21/2006	No comma added to separate keywords	multiple classes: `EntryEditor`, `GroupDialog`, `BibtexParser`, `parseFieldContent`
T3	1594123	11/10/2006	Failure to import big numbers	single method: `BibtexParser.parseFieldContent`
T4	1489454	5/16/2006	Acrobat Launch fails on Win98	single method: `Util.openExternalViewer`

4.3.4 iTrace

For capturing eye-tracking data and linking it to source code elements in the IDE, we developed and use a new version of our Eclipse plugin iTrace [Shaffer et al., 2015]. For this new version, we added the ability to automatically and on-the-fly link eye gazes to fine-grained AST source code elements, including method calls, variable declarations and other statements in the Eclipse IDE. In particular, iTrace gives us the exact source code element that was looked at with line-level granularity. Furthermore, to support a more realistic work setting, we added features to properly capture eye gazes when the developer scrolls or switches code editor windows in the IDE, or when code is edited. Eye-tracking on large files that do not completely fit on one screen is particularly challenging as none of the state-of-the-art eye-tracking software supports scrolling while maintaining context of what the person is looking at. Our new version of iTrace overcomes this limitation and supports the collection of correct eye gaze data when the developer scrolls both, horizontally and vertically as well as when she switches between different files in the same or different set of artifacts.

iTrace interfaces with an eye-tracker, a biometric sensor usually in the form of a set of cameras that sit in front of the monitor. For our study, we used the Tobii X60 eye-tracker [Tobii, 2015] that does not require the developer to wear any gear. Tobii X60 has an on-screen accuracy of 0.5 degrees. To accommodate for this and still have line-level accuracy of the eye gaze data, we chose to set the font size to 20 points for source code files within Eclipse. We ran several tests to validate the accuracy of the collected data.

After calibrating the eye-tracker through iTrace's calibration feature, the developer can start working on a task and the eye gazes are captured with the eye-tracker. iTrace processes each eye gaze captured with the eye-tracker, checks if it falls on a relevant UI widget in Eclipse and generates an eye gaze event with information on the UI in case it does. iTrace then uses XML and JSON export solvers, whose primary job is to export each gaze event and any information attached to it to XML and JSON files for later processing.

Currently, iTrace generates gaze events from gazes that fall on text and code editors in Eclipse. These events contain the pixels X and Y coordinates relative to the top-left corner of the current screen, the validation of the left and right eye as reported by the eye-tracker (i.e., if the eye was properly captured), the left and right pupil diameter, the time of the gaze as reported by the system and the eye-tracker, the line and column of the text/code viewed, the screen pixel coordinates of the top-left corner of the current line, the file viewed, and if applicable, the fully qualified names of source code entities at the gaze location. The fully qualified names are derived from the abstract syntax tree (AST) model of the underlying source code. For this study, we implemented iTrace to capture the following AST elements: classes, methods, variables, enum declarations, type declarations, method declarations, method invocations, variable declarations, any field access, and comments. These elements are captured regardless of scope, which includes anonymous classes.

4.3.5 Data Collection

For this study, we collected data on participants' eye traces and their interactions with the IDE simultaneously. Since we conducted our study with the Eclipse

IDE, we used the Eclipse plugin Mylyn [Mylyn, 2015, Kersten and Murphy, 2005] that monitors a user's interactions with code elements in the IDE, in particular *selects* and *edits* of classes, methods and fields. For the eye-tracking data, we used our new version of the Eclipse plugin iTrace [Shaffer et al., 2015]. In our analyses, we only considered edit and selection interaction events and eye gazes on java files and did not analyze captured data on other file types, such as html or xml files.

We gathered a total of 66 change task investigations from the 12 professional developers and 10 computer science students who each worked on three different change tasks. For each of these investigations, we gathered the eye-tracking data and the user interaction logs. Due to some technical difficulties, such as a participant wearing thick glasses or too many eye gazes not being valid for a task, we excluded 11 change task investigations and ended up with 55 overall: 18 subjects investigating change task T2, 16 subjects investigating change task T3, and 21 subjects investigating change task T4. These 55 change task investigations comprise totally 119,618 single eye gazes and 3524 single interactions with methods, classes and fields. With respect to individual method investigations over all participants and tasks, we gathered a total of 688 method investigation instances.

4.4 Study Results

Based on the collected gaze contexts and interaction contexts of the 22 participants we were able to make detailed observations on how developers navigate within source code and build up their contexts. Table 4.2 presents some descriptive statistics over the gathered interaction and gaze context averaged over all participants and change tasks, and Table 4.3 presents data on the gaze and interaction context per participant. The data presented in these tables already highlights the often big differences between the elements and events captured in the different kinds of context—interaction and gaze—as well as the very fractional reading of methods for developers' change task investigations.

In the following, we will further discuss this data in more detail and in the context of the concrete observations we made. We structure our *observations* along four research foci: the difference between gaze and user interaction data, developers' navigation within methods, developers' navigation between methods and developer-specific navigation characteristics and start each paragraph with the observation we made. Since almost all participants used the maximum time of twenty minutes for the change task investigations, we did not perform any analysis of the data with respect to task completion time.

Table 4.2: Descriptive statistics of the analyzed interaction and gaze contexts gathered for the change tasks, averaged over all participants and tasks (± denotes the standard deviation).

Variable	Description	Interaction	Gaze
Num_{UMe}	Number of unique methods which were selected, edited, or looked at	6.0 (±4.5)	12.5 (±11.8)
Num_{MeSw}	Number of times a developer selected or looked at a different method	5.8 (±5.2)	73.5 (±78.5)
$\text{Rat}_{\text{SwInVsAll}}$	Ratio between the number of switches to a method within the same class and the number of method switches	54.4% (±33.8)	88.3% (±14.1)
$\text{Rat}_{\text{SwOutVsAll}}$	Ratio between the number of switches to a method in a different class and the total number of method switches	45.6% (±33.8)	11.7% (±14.1)
$\text{Rat}_{\text{ProximitySw}}$	Ratio between the number of switches to a method right above or underneath and the total number of method switches within a class	69.9% (±39.0)	37.0% (±25.6)
Dwell_{Me}	Time spend reading a method		0.3min (±0.5)
$\text{Num}_{\text{LineSw}}$	Number of line switches within a method		40.0 (±101.0)
$\text{Perc}_{\text{LinesLooked}}$	Percentage of lines which were looked at within a method		32.2 (±25.0)
$\text{Num}_{\text{Me>HfLi}}$	Number of methods for which a developer looked at more than half of the lines		7.2 (±10.1)
$\text{Num}_{\text{UMe>HfLi}}$	Number of unique methods for which a developer looked at more than half of the lines		2.7 (±2.9)
$\text{Tsp}_{\text{ToMe>HfLi}}$	Time span from start of change task investigation to first method for which a developer looked at more than half of the lines		2.9min (±2.9)
$\text{Num}_{\text{Me>AvgDw}}$	Number of methods a developer spent more than $Dwell_{Me}$ (average time reading a method)		13.6 (±16.0)
$\text{Num}_{\text{UMe>AvgDw}}$	Number of unique methods a developer spent more than $Dwell_{Me}$		5.0 (±4.8)
$\text{Tsp}_{\text{ToMe>AvgDw}}$	Time span from start of change task investigation to first method for which a developer looked at for more than $Dwell_{Me}$		2.1min (±2.1)

Table 4.3: Summary of professional (pro) and student (stu) developers' average (avg) of methods and method switches captured in the gaze and interaction context over all three change tasks, as well as the average percentage of lines read within methods.

ID	avg # of method switches		avg # of unique methods		% of lines
	gaze context	inter. context	gaze context	inter. context	looked at
P1	6.5	3	4.5	3.5	31.7%
P2	59.7	10	12	8	32.4%
P3	50	7.5	15	8	23.6%
P4	46	3.5	16.5	3.5	32.9%
P5	126	12.5	14	10.5	25.8%
P6	22.5	4.5	5.5	5.5	47.0%
P7	226	8.7	39.3	8.7	35.0%
P8	47.7	3	5.3	4	26.9%
P9	50.5	3	6.5	4	41.4%
P10	172	9	9	8	71.4%
P11	64	6.7	12.3	6	30.2%
P12	138	5	8	6	45.4%
avg pro	83.73	6.42	13.38	6.46	33.6%
S1	13.3	2	8.7	3	28.4%
S2	20	1.7	6.7	2.3	24.7%
S3	45.3	2.4	8.7	3.3	27.3%
S4	96.3	15	23.7	14.7	35.5%
S5	96	7	11.7	7.6	37.4%
S6	10.5	3.5	3	4.5	19.4%
S7	142.3	0.7	9	1.7	34.5%
S8	64	4.7	19.7	5.3	25.1%
S9	59.7	5	8.3	4.3	33.3%
S10	77	9	15	9.3	28.5%
avg stu	64.24	5.14	11.72	5.66	30.6%
avg total	**73.45**	**5.75**	**12.51**	**6.04**	**32.16%**

4.4.1 Interaction Context and Gaze Context

O1—Gaze contexts capture substantially more, and more fine-grained data.

To compare the different amounts of elements within the gaze and the interaction contexts, we used a paired-samples t-test[1] with pairs consisting of the gaze and the interaction context for a task and subject.

This paired-samples t-test showed that the number of different classes contained in the gaze context (Mean (M) = 4.78, Standard Deviation (SD) = 3.58) and the number of different classes contained in the interaction context ($M = 4.42, SD = 3.00$) do not differ significantly ($t(54) = 1.98, p = .053$). Nevertheless, there were more classes captured in the gaze contexts, which turned out to be internal classes or classes defined in the same file. While there is no significant difference on a class level, there is a significant difference in the amounts of methods captured. The number of different methods within the gaze contexts ($M = 12.51, SD = 11.75$) is significantly higher than the number of different methods within the interaction contexts ($M = 6.04, SD = 4.53$), $t(54) = 4.57, p < .05$). This observation on the substantial difference in the number of elements within the gaze and interaction context provides evidence that developers often look at methods that they do not select. Approaches that only analyze interaction logs, thus miss a substantial amount of information.

When analyzing the method sequences captured in the logs, the data also shows that gaze context not only captures more elements, but also more details on the actual sequences of navigation between methods. A paired-samples t-test revealed a significant difference in the number of method switches captured in gaze contexts ($M = 73.45, SD = 78.47$) and the number of method switches captured in interaction contexts ($M = 5.75, SD = 5.17$), $t(54) = 6.52, p < .05$. Table 4.3 summarizes the number of unique methods and the number of method switches for each context type and participant.

[1]According to the central limit theorem, with large samples number (>30), the distribution of the sample mean converges to a normal distribution and parametric tests can be used [Field, 2005].

O2—Gaze and Interaction Contexts capture different aspects of a developer's navigation.

To evaluate whether gaze and interaction contexts capture different aspects of a developer's navigation for change task investigations, we defined ranking models based on the data available in the different contexts and compared the top ranked methods. There are a variety of models that can be used to select the most important elements within a navigation sequence [Piorkowski et al., 2011]. For our analysis, we used single-factor models to select the most important elements in each kind of context that were also suggested in previous studies [Parnin and Gorg, 2006, Piorkowski et al., 2011]. To rank the methods of a gaze context we used a time-based model. This model ranks methods higher for which a developer spends more time looking at. To rank the methods of an interaction context we used a frequency-model, which ranks methods higher that were visited more often.

We compared for each change task investigation the top 5 methods resulting from the frequency model and from the time-based model. We then analyzed for how many methods the interaction and the gaze context agree and found that in 65.03% ($SD = 32.26\%$) of the recommended methods this was the case.

Comparing solely the highest ranked method for each context pair results in an agreement of 27.27%. The agreement on the top 5 most important methods however is considerably lower for change task T2 ($M = 52.31\%, SD = 34.98\%$) than for change task T3 ($M = 71.88\%, SD = 27.62\%$) and for change task T4 ($M = 70.71\%, SD = 31.32\%$). While the description for change task T3 and change task T4 include concrete hints to source code elements which are possibly important for performing the change task, change task T2 required to explore the source code more exhaustively in order to find the relevant code and a possible fix. These results illustrates that gaze context, especially in form of the time of gazes, captures aspects that are not captured in the interaction context and that might be used to develop new measures of relevance. Especially, since gaze contexts also capture elements that are not in the interaction context (**O1**), the more fine-grained gaze data might provide better and more accurate measures of relevance.

4.4.2 Within Method Navigation

We base the analysis of navigation within methods solely on the gaze data, since interaction contexts do not capture enough detail to analyze within method navigation.

O3—Developers only look at few lines within methods and switch often between these lines.

Figure 4.1 depicts the lines of a randomly chosen professional developer (middle) and a randomly chosen student developer (right) looked at within a certain method and over time during a change task investigation.

Across all subjects and tasks, developers only look at few lines within a method, on average 32.16% ($SD = 24.95\%$) of the lines. The lengths of methods included in this analysis thereby differed quite a lot, with an average length of 53.03 lines ($SD = 139.37$), and had a moderate influence on the number of lines looked at by a developer, Pearson's $r = .398, p = .01$.

Participants performed on average 39.95 ($SD = 100.99$) line switches within methods. The method length again influences the amount of line switches moderately, Pearson's $r = .305, p = .01$.

Further examination of the kind of lines developers actually looked at shows that developers spend most of their time within a method looking at method invocations ($M = 4081.98ms$) and variable declaration statements ($M = 1759.6$ ms), but spent surprisingly little time looking at method signatures ($M = 1090.67$). In fact, in 319 cases out of 688 method investigations analyzed, the method signature was ignored and not looked at. Our findings demonstrate that developers who are performing an entire change task involving several methods and classes, read methods differently than developers who are reading methods disconnected from any task or context, in which case the method signature might play a stronger role.

O4—Developers chase data flows within a method.

To better understand how developers navigate within a method, we randomly picked six change task investigation instances from the collected gaze contexts and

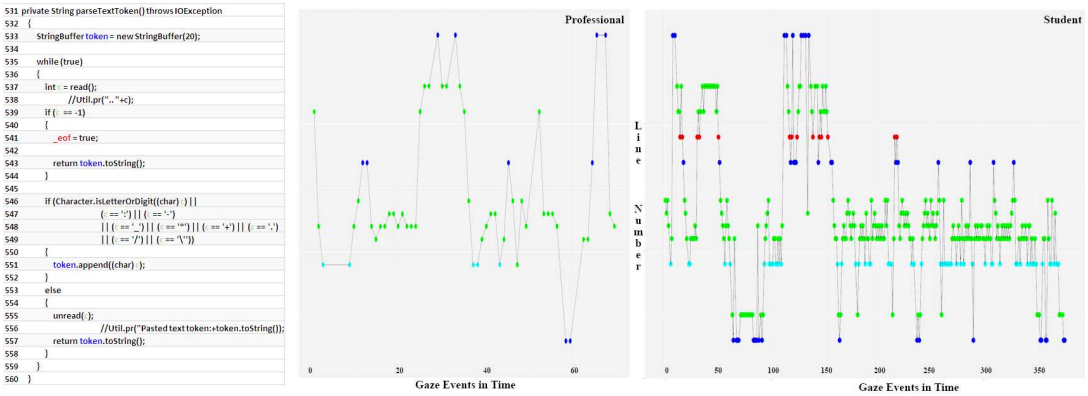


Figure 4.1: The sequence logs mapped to line numbers and colors, with the colored source code on the left.

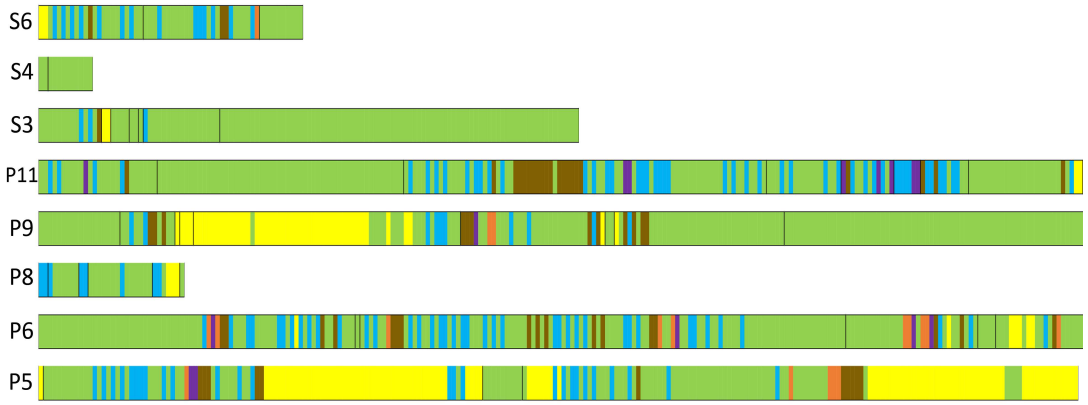


Figure 4.2: Colored sequence logs of eight participants investigating method `BrowserLauncher.locateBrowser`. Each row represents a method investigation of a participant with the time axis going from left to right. Eye gazes on lines that talk about the same variable are colored the same. For instance, S3 exclusively gazed at lines containing the same variable for more than the second half of the method investigation, and thus more than the second half of the bar for S3 is colored green.

manually retraced the paths participants followed through a method by drawing their line switches on printouts of the methods. Closely examining these printed methods with the eye traces drawn on top, allowed us to form the observation that developers often trace variables when reading a method. To investigate

this observation further, we selected four methods which were investigated by most participants, resulting in 40 unique method investigation instances. The 40 method investigation instances stem from 18 different participants and two different tasks. 22 of these 40 investigations stem from professional software developers, while the other 18 stem from students.

For each method, we assigned a color to each variable used within the method. Colors were chosen randomly. We then colored the lines in which a variable was either defined or used in the method. We did not color lines or statements that did not include a variable. In case more than one variable was used in a single line, we manually checked if a color was predominantly used before or after the line was visited and used the predominant color. In cases where there was no evidence of a predominant color, we picked the color of the variable that was used first in the source code line. Over all four methods, we identified an average of 7.25 variable slices per method with an average of 6.2 different lines of code per slice, i.e., an average of 6.2 lines in a single method referred to the same variable.

In a next step, we took participants' eye gaze logs—the sequentially ordered eye gaze events including line numbers—for these four methods, filtered the gaze events that did not map to a variable slice, such as brackets and empty lines, and then colored each log entry with the colour of the variable it referred to. Figure 4.2 illustrates some of these color-coded eye gaze logs from participants' method investigations with the sequence of events going from left to right.

Our analysis revealed that developers switched between the lines of these four methods on average 178.0 ($SD = 189.9$) times. We then used our color coding to examine how many of these line switches are within a variable slice, i.e., lines that refer to the same variable and have the same color. Over all method investigation instances we found an average of 104.2 (112.1) line switches of the 178 to be within a variable slice, supporting our observation that developers are in fact following data flows when investigating a method. The long green and yellow blocks within Figure 4.2 provide further visual evidence on the high frequency of participants switching between lines within a variable slice (same color) rather than switching between different variable slices (different consecutive colors).

4.4.3 Between Method Navigation

Overall, subjects switched on average 73.45 ($SD = 78.48$) times between methods when working on a change task. Thereby, they revisited a method on average 5.44 times.

> ***O5—Developers frequently switch to methods in close proximity and rarely follow call relationships.***

To investigate the characteristics of method switches we examined whether they were motivated by call relationships or due to the close proximity of methods. We assessed for each method switch within a class and for each method switch to a different class whether the switch was motivated by following the call graph of the method. In addition, we assessed for each method switch within the same class whether the sequentially next method looked at is directly above or directly below the current method. We conducted this analysis for both contexts: the gaze context and the interaction context.

To understand if a method switch was motivated by following the call graph we memorized the method invocations within a given method and assessed if the next method in the method sequence was one of the memorized invoked methods. While we had to consider all method invocations within a given method when analyzing the interaction context, we could precisely assess at which method invocation the developer actually looked at when analyzing the gaze context. If a next method in the sequence was equal to one of the memorized invoked methods, we concluded that it is likely that the developer followed the call relationship (switch potentially motivated by call graph), although, the next method could have also been within spatial proximity and the call relationship not of importance for the navigation. If the next method was not contained within the memorized method invocations we concluded that the developer's navigation was motivated by other means than the call relationships. To understand if a method which was looked at next is directly above or directly below a current method, we compared the line numbers in the source file.

Gaze context. We found that merely 4.05% ($SD = 6.68\%$) of all method switches were potentially motivated by following the call graph. On average,

the subjects switched methods potentially motivated by the call graph more when they were investigating change task T4 ($M = 6.57\%, SD = 9.36\%$) than when they were investigating change task T2 ($M = 1.87\%, SD = 2.94\%$) and change task T3 ($M = 3.18\%, SD = 4.34\%$). A paired-samples t-test showed that developers switched methods potentially motivated by the call graph significantly more often within a class ($M = 4.44\%, SD = 7.12\%$) than between different classes ($M = 0.70\%, SD = 4.50\%$), $t(54) = 3.17, p = .003$.

At the same time, a larger amount of all method switches ended in methods which were right above or below a method ($M = 36.95\%, SD = 25.57\%$). These results suggest that the call graph of a project is not the main drive for navigation between methods, but the location of a method captures an important aspect for navigation between methods.

Interaction context. We found that 22.61% ($SD = 29.09\%$) of all method switches were potentially motivated by following the call graph. Different to the results of the gaze context analysis, participants switched between methods potentially motivated by the call graph substantially more when they were investigating change task T3 ($M = 38.23\%, SD = 31.56\%$) than when they were investigating change task T2 ($M = 8.05\%, SD = 13.89\%$) and change task T4 ($M = 23.19\%, SD = 31.42\%$). On average, subjects followed considerably more call relations when they were navigating within the class ($M = 24.15\%, SD = 34.71\%$) than when they were navigating to a method implemented in another class ($M = 6.44\%, SD = 20.74\%$).

We further found that on average 69.93% ($SD = 39.01\%$) of the method switches within a class were aimed towards methods which are directly above or below a method.

Overall, these results also show that the more coarse grained interaction context indicates that developers follow structural call graphs fairly frequently (22.6%) while the more fine grained gaze context depicts a different picture with only 4.1% of the switches being motivated by structural call relations. To understand whether these switches to methods in close proximity were intentional or mainly present an inadvertent glimpse to a neighbouring method, we examined

how many lines of the neighbouring method a developer looked at. We found that in 30.20% of the method switches to a method in close proximity were rather an inadvertent glimpse with the developer only looking at a single line of the method, while in 36.45% of the cases the developer read the nearby method more carefully, i.e., she read more than half of the lines of the method. This indicates that a big part of the switches to proximate methods serves a purpose and is not necessarily caused by inadvertently wandering around.

Our results on switches to methods in close proximity further support the findings of a recent head-to-head study that compared different models of a programmer's navigation [Piorkowski et al., 2011] and that suggested to use models to approximate a developer's navigation based on the spatial proximity of methods within the source code.

O6—Developers switch significantly more to methods within the same class.

Applying a paired-samples t-test on the gaze context shows that developers switched significantly more between methods within the same class ($M = 65.22, SD = 73.20$) than they switched from a method to a method implemented in another class ($M = 8.24, SD = 11.95$), $t(54) = 6.07, p < .001$. While, over all three tasks, participants rarely switched to methods of different classes, the participants' method switching within the same class differs between tasks. A Wilcoxon matched pairs signed rank test indicates that participants switched significantly more between methods within classes for change task T2 ($M = 103.50, SD = 106.23$) than for change task T4 ($M = 36.31, SD = 39.08$), $z = -2.66, p = .008$. While it is not surprising that different tasks result in different navigation behavior of participants, this also suggests that it is important to take into account the task for support tools, such as code navigation recommendations.

O7—Developers only read few of the explored methods more thoroughly during a change task investigation.

While developers read parts of several methods for each task, they only read very few of these methods more thoroughly. In only 24.54% ($SD = 19.36$) of the

unique methods that developers explored, they spend time to read more than half of the lines of the method see Table 4.3). Professional developers thereby read on average more methods more thoroughly ($M = 27.17\%$, $SD = 16.18\%$) than student developers ($M = 22.18\%$, $SD = 21.83\%$), although this difference is not statistically significant.

For all change tasks, there is also only little overlap amongst the developers with respect to the more thoroughly explored methods, i.e., different developers explored different methods more thoroughly. For instance, while there was one method that 16 of the 21 participants that worked on task T4 explored more thoroughly, for all other methods that were explored more thoroughly for this task, it was only an average of 3.38 of the 21 participants doing so. For task T2 it was even just an average of 1.48 developers of the 18 working on this task that explored the same method more thoroughly.

To see whether a method's complexity might have an influence on the number of developers reading a method more thoroughly, we looked at McCabe Cyclomatic Complexity. Our analysis showed that, for instance for task T4, the complexity scores of the methods that developers read more thoroughly do not correlate with the numbers of developers who focused on a more thoroughly read method (Pearson's $r = .077, p = .768$). Further investigations on what the reason might be for methods being read more carefully is planned for future work.

4.4.4 Developer-Specific Context Characteristics

Studies showed that the source code elements captured in task contexts are highly developer-specific [Fritz et al., 2014b]. To train an individual navigation recommender tool, a rather large history of interaction data is needed, which is often unavailable. Hence, we aim to explore whether we can identify different groups of developers that explore source code in a similar way, such that recommendation tools might be adapted to groups rather than individuals. Further, we explore how the context of developers who successfully solved a change task differs from the contexts of developers which have not had enough time to complete the change task. Finally, we also look into how developers with a rich programming

experience build up context compared to developers with less programming experience.

O8—Developers either use a skimming strategy or a seeking strategy to explore source code for a change task.

To investigate whether our data includes different groups of developers, which explore source code in a similar way, we conducted a cluster analysis on the gathered gaze contexts. We used an agglomerative hierarchical clustering algorithm, as we followed a more exploratory approach and did not want to decide on the number of clusters beforehand. We used the log-likelihood as distance measure. In this analysis we focused on the gathered gaze contexts, as the data collected through interactions is too coarse-grained to detect specific code exploration strategies.

We obtained two clusters from our analysis. These clusters have a silhouette measure of cohesion and separation of 0.5, which denotes a good cluster quality (a silhouette measure below 0.2 denotes poor cluster quality, while a silhouette measure of 0.5 and higher denotes a good cluster quality). The two clusters have different sizes. The first cluster comprises 34.5% of the data points, while the second cluster comprises 65.5% of the data points. The variables which influence the classification are the ratio of switches to a method right above or below, the average percentage of lines which were looked at within methods, and the number of methods on which developers spent more than their average method investigation time.

Based on the values for these variables, we interpret one cluster as "seeking" the source code and the second cluster as "skimming" the source code. Code exploration instances belonging to the "seeking" cluster are characterized by less switches to proximate methods ($M = 0.29$), more lines being read within a method ($M = 0.37$), and by having considerably more focus points than the code exploration instances in the "skimming" cluster, with an average of 16.97 methods. Skimming the source code on the other hand is characterised by more switches between proximate methods ($M = 0.67$), reading less lines within methods ($M = 0.22$), while focusing on average at far less methods than seekers

with an average of 7.26 methods. Overall, the analysis suggests that developers seek source code more often (36 of the task investigations were classified into this group) than they skim the source code (19 of the task investigation sessions were classified into this group).

Looking at these clusters, we recognized that the change task itself has a high impact on a developer's code exploration behavior, i.e. whether the developer seeks or skims the source code. We discovered that all developers, except for one, were seeking the source code when investigating change task T3 (see Table 4.1). Change task T3 is the only change task included in our analysis which has a stacktrace. As the stacktrace offered more code specific information about the change task to the developers, almost all of them applied a seeking-strategy. These results empirically show that the information given in change tasks can influence the way how developers build contexts. This further confirms the survey results by Bettenburg et al. [Bettenburg et al., 2008] that showed stacktraces in change tasks are among the most important information fragments for solving a change task.

17 developers investigated both of the remaining change tasks T2 and T4. Considering only the change task investigations for these two change tasks, our clustering method could assign 65% of these 17 developers to one specific category. 6 developers were identified as seekers, while 5 developers were identified as skimmers. The remaining 6 developers applied each time another strategy for these two change tasks.

O9—Developers who solved a change task successfully read more lines within methods, switched more between these lines, and focused on more methods.

How does a context leading to a successful change look like? While each developer's context is very individualized, we explore whether contexts which allowed for a successful change have commonalities. Exploring commonalities of successful or faster changes might inform new tool support, which directs developers to adopt more efficient navigation behavior.

Based on the changes made by the developers and the short descriptions of their solutions, we manually assessed whether the study participants successfully

solved the given change tasks. Over all 55 change task investigations we determined that 12 change tasks were solved successfully. Each of the three change task types is among the successfully solved ones, although change task T3 was most often (6 times) solved successfully.

Since almost each change task investigation referring to change task T3 was classified as a seeking-session (see **O8**), most of the successful change task investigation are classified as seeking-sessions (75%). We ran a Mann-Whitney U test to compare the different variables related to the gathered gaze contexts and the gathered interaction context (see Table 4.2) for successfully solved change tasks and unsuccessfully solved change tasks. For the gaze context our analysis suggests that developers who solved a change task successfully switched significantly more between lines when reading a method ($U = 119, p = .005$). The amount of lines they looked at on average when reading a method and the number of methods they focused on are also considerably different, although not significant ($U = 165, p = .058$, respectively $U = 163, p = 0.052$). However, since change task T3 had a stacktrace included, we ran the same analysis on the dataset excluding change task T3. The results suggest the same parameters to be decisive. Developers who successfully solved change task T2 and T4 switched on average more between lines of a method ($U = 33, p = .008$), focused on more methods ($U = 57.5, p = .012$) and read more different lines within a method ($U = 49, p = .52$). Since too few change task investigations for change task T2 and T4 were solved successfully (2 for change task T2 and 4 for change task T4), we cannot conclude whether the choice of strategy has an impact on the task outcome. We plan to investigate whether a particular investigation strategy has an impact on the task outcome in future studies.

When analyzing the interaction contexts with respect to successful and unsuccessful changes, we did not find any significant differences.

> ***O10—There were no significant differences in contexts built by professional developers and student developers in our study.***

Previous empirical studies on software developers found differences in the patterns that experienced and novice developers exhibit (e.g., [Crosby and

Stelovsky, 1990]). To investigate such differences, we analyzed our data for differences in navigation between our professional developers and our students. In particular, we tested each statistic that contributed to the above observations and examined whether there were any statistically significant differences in gaze, respectively interaction contexts. To compare the professional developers and the students we used a Mann-Whitney test, as there are different participants in each group and the data does not meet parametric assumptions. Overall, we did not find any statistically significant difference between the two groups of participants in the amounts of unique elements on different granularity levels within the gaze context ($U = 341.0, p = .539$ on class level, $U = 363.5, p = .820$ on method level) nor the interaction context ($U = 368.0, p = .878$ on class level, $U = 286.5, p = .125$ on method level). Furthermore, there was no significant difference in the amounts of switches conducted between different elements within a class ($U = 314.5, p = .292$ for the gaze contexts and $U = 297.5, p = .174$ for the interaction contexts) nor outside of a class ($U = 337.0, p = .495$ for the gaze contexts and $U = 266.5, p = .058$ for the interaction contexts). Finally, we also could not find any significant difference in the amount of call relationships followed ($U = 325.5, p = .362$ for the gaze contexts and $U = 268.0, p = .055$ for the interaction contexts) nor if any of these two groups switched more often to methods with a high spatial proximity ($U = 367.5, p = .873$ for the gaze contexts and $U = 332.0, p = .445$ for the interaction contexts). So even though our exemplary figure (Figure 4.1) that depicts a sequence log for a professional and a student developer might suggest a difference in navigation behavior, our analysis did not produce any such evidence.

4.5 Approaches

In this section we demonstrate the potential of fine-grained task context on the basis of two approaches.

4.5.1 Fine-Grained Navigation Recommendation

When developers explore source code, they navigate extensively between methods. To support developers during the time-consuming source code navigation, different approaches have emerged (e.g., [DeLine et al., 2005b, Zimmermann et al., 2005]). These approaches are based on an underlying model which imitates developers' navigation steps and points the developers directly to interesting places to go to next. A head-to-head study by Piorkowski et al. [Piorkowski et al., 2011] compared a variety of underlying models, and found that recently visited methods and methods which are in close proximity are most likely to be visited next.

Using the gathered gaze contexts, we transferred these approaches to the much finer-granular line navigation within methods. Specifically, we evaluated four models based on our observations (see Section 4.4) and previous research to predict the next source code line a developer will visit. Based on observation **O3** which states that developers only look at a few lines within a method and switch often between these lines, we formulated a recency- and frequency-based model. Based on the observation **O4** which states that developers chase data flows within methods we formulated a data flow-based model. Finally, inspired by our observation **O5** which states that developers frequently switch between methods in close proximity, we also included a proximity-based model in this experiment. In summary, we included the following within method navigation models in our experiment:

- *Data flow-based model*: ranks the source code lines within a method higher that include a variable occurring in the current line of focus.

- *Proximity-based model*: ranks the source code lines higher, which are in close proximity to the current line of focus. In case of an uneven number of recommendations, this model favors the lower part of the method, as more methods are read strictly from top to bottom (64.77%).

- *Recency-based model*: ranks the source code lines higher that were looked at more recently.

- *Frequency-based model*: ranks the source code lines higher that were looked at more frequently.

Similarly to the evaluation strategy applied by Piorkowski et al. [Piorkowski et al., 2011], we calculated each model's average accuracy for the top-N results. In this analysis we tested results using an N value which ranges from 1 to 10. We calculated the hit ratio for each method investigated by each developer by comparing the top N results produced by each model with each next line visited by the developer. The hit ratios averaged over all subjects and methods investigated are depicted in Figure 4.3 and can be summarized in the following *finding*:

F1*—*The proximity-based model has the highest hit ratio over all prediction models for each $N \leq 10$***.***

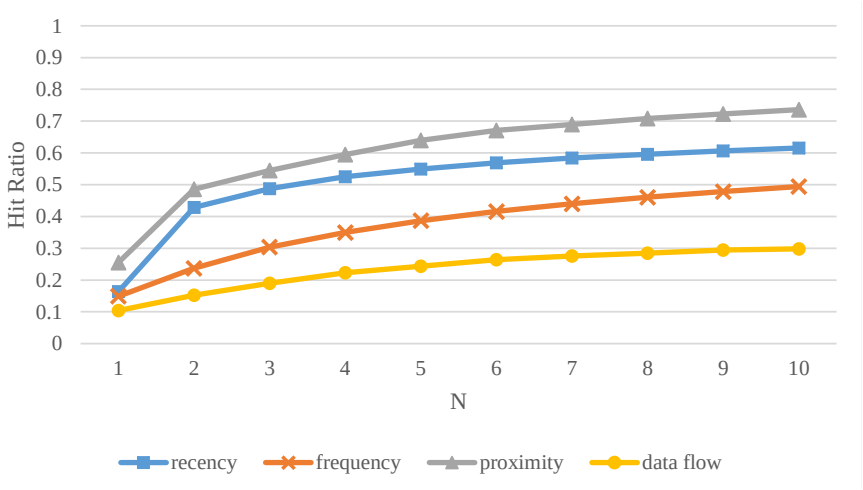


Figure 4.3: The averaged hit ratios over all subjects and methods investigated.

Overall, the proximity-based model converges most to the developers' navigation within methods. However, the average length of the methods included in this analysis differs quite a lot ($M = 53.03$, $SD = 139.37$). Thus, in the case of a short method (method length <= N) the proximity-based model simply

recommended all the lines of a method and hence caused a high accuracy. The relatively high accuracy of the recency- and frequency-based models confirms our observation **O3** that states that developers switch a lot between lines they already visited.

Due the comparatively low accuracy of the data flow-based model that does not provide strong support for **O4**, we conducted a follow up analysis to investigate possible causes. For this analysis, we grouped the methods investigated into "focused" and "skimmed" methods as we did when exploring **O7**. Methods of which more than half of the lines were looked at are defined as "focused", while the remaining ones are defined as "skimmed" methods. We then again calculated the accuracy of each model for just the "focused" methods. The results are depicted in Figure 4.4 and can be summarized in the following *finding*:

> ***F2—The hit ratios of the data flow-based model are substantially higher for "focused" methods compared to all explored methods.***

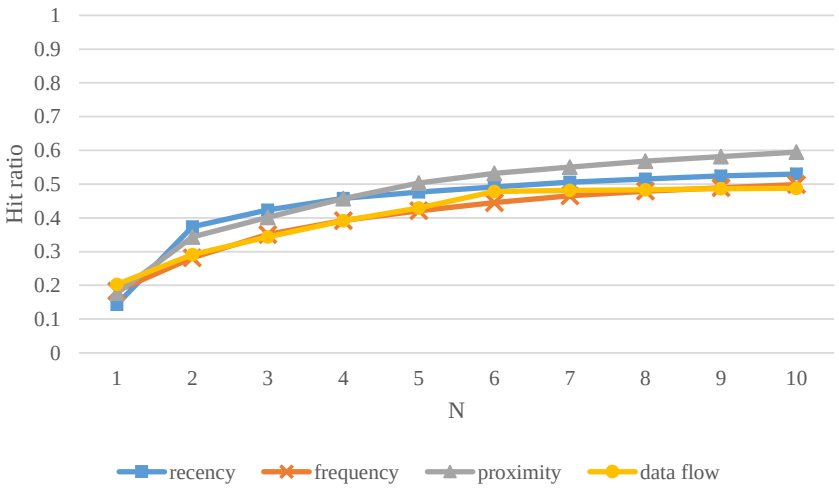


Figure 4.4: Average hit ratios of each prediction model overall focused methods.

When looking at the top recommendations of each model applied to the group of focused methods ($N = 1$), the data flow-model outperforms the remaining

models with a hit ratio of 0.2. Overall, the different models do not differ as much for the focused methods as they do when applying the line prediction models to all methods (see Figure 4.3). In particular, the proximity-based model's hit ratios are considerably lower.

These results further show that it is highly important to understand which methods developers focus on when investigating a change task and that an automatic detection thereof could be of high value for tool support.

4.5.2 Predicting Task Difficulty

Captured task contexts can be used to support different software engineering steps and aspects, for example to enable a task-focused development environment [Kersten and Murphy, 2006, Bragdon et al., 2010], to support code navigation [DeLine et al., 2005b], or to localize reported bugs in the source code [Kevic and Fritz, 2014]. While many of these approaches are based on the source code elements within the captured contexts, we additionally investigate the sequence of gaze and navigation steps.

In particular, we use the gathered interaction and gaze contexts to predict perceived task difficulty of a developer when working on a change task. Knowing whether a developer experiences difficulties when working on a change task might inform approaches for prioritizing change task reviews or even for better interruption management.

For this analysis, we used the study participants' ranking of the change tasks into one of three categories of perceived difficulty and applied a stepwise multinomial logistic regression to predict membership in one of these categories. We used a stepwise multinomial logistic regression [Field, 2005] on the variables gathered for the interaction and gaze contexts (see Table 4.2). We think that these parameters represent a good starting point for our analysis and we are also not aware of any previous research in the area that already identified variables for this kind of prediction in this context.

Of the 55 change task investigations, participants ranked 8 (14.5%) as "easy", 26 (47.3%) as "average", and 21 (38.2%) as "difficult". Each task difficulty category includes change task investigation instances for all three kinds of change

tasks of our study (see Table 4.1). Also, only change task investigations with the "easy" and "average" difficulty were performed successfully, providing further support for the validity of the rankings.

The final model of both kinds of context—interaction and gaze—allowed us to predict the perceived task difficulty significantly better than with the baseline model, i.e. a model that omits all variables and only uses the constant (see Table 4.4). The final prediction model based solely on gaze context, which captures substantially more, and more fine-grained data than interaction context (**O1**), allows a higher decrease in unexplained variance from the baseline model to the final model ($\chi^2(4) = 20.44, p < .001$), than the final prediction model based on the interaction context ($\chi^2(2) = 7.57, p = .023$). For the interaction context as well as for the gaze context the switch ratio to methods in close proximity has a significant effect on predicting the perceived task difficulty. For the interaction context, the switch ratio to methods in close proximity is significant with a p-value of $p = .023$ and decreases the amount of unexplained variance by 7.57. For the gaze context, the time to first focus ($\chi^2(2) = 8.88$,$p = .012$) and the switch ratio to methods in close proximity ($\chi^2(2) = 11.56$,$p = .003$) significantly helps to predict the difficulty level.

The parameter estimates that allow to compare two categories with each other (e.g., how the parameters compare for "easy" to "difficult") are summarized in Table 4.5.

Gaze context. The parameter estimates presented in Table 4.5 indicate that for tasks that are perceived as "easy", developers skim the source code longer before a method was read thoroughly, i.e. read more than half of its lines, and that they looked less frequently to methods in close proximity than for tasks that are perceived "average" or "difficult".

Interaction context. The parameter estimates presented in Table 4.5 paint a different picture for the navigation behavior when it is used to predict perceived difficulty. In particular, the parameters indicate that for tasks that are perceived as "average" developers are more likely to *select* methods in close proximity more often than for tasks that are perceived as "difficult".

Table 4.4: Results of applying a multinomial logistic regression to parameters of the gaze and the interaction context (2 Log-Likelihood measures how much unexplained variability there is, χ^2 = chi-square, df = degrees of freedom, Sig. = Statistical significance).

	2 Log-Likelihood	χ^2	df	Sig.
interaction context				
baseline model	45.06	-	-	-
final model	37.48	7.57	2	.023
proximate switches ratio	37.48	7.57	2	.023
gaze context				
baseline model	110.26	-	-	-
final model	89.81	20.44	4	$< .001$
time to first focus	101.37	8.88	2	.012
proximate switches ratio	89.81	11.56	2	.003

F3—The more often a developer looks at methods in close proximity and also the less time it takes until a developer explores a first method thoroughly, the more likely the developer perceives the task as difficult.

4.6 Threats to Validity

One threat to validity is the short time period each participant had for working on a change task. Unfortunately, we were limited by the time availability of the professional developers and therefore had to restrict the main part of the study to one hour. While the data might thus not capture full task investigations, it provides insights on investigations for multiple change tasks and thus the potential of being more generalisable. Furthermore, as participants were investigating three change tasks in the same source code, there might be a learning effect

Table 4.5: Parameter estimates of applying a multinomial logistic regression to the gaze and the interaction context (B = coefficient, Std.Error = Standard Error, df = degrees of freedom, Sig. = Statistical significance).

		B	Std.Error	df	Sig.
gaze context	*"average" vs. "easy"*				
	Intercept	.05	.78	1	.95
	proximate switches ratio	8.13	3.63	1	.025
	time to first focused method (time-based)	-.106	.041	1	.01
	"difficult" vs. "easy"				
	Intercept	-.56	.86	1	.518
	proximate switches ratio	9.37	3.73	1	.012
	time to first focused method (time-based)	-.15	.05	1	.005
interaction context	*"easy" vs. "difficult"*				
	Intercept	-1.62	.83	1	.05
	proximate switches ratio	1.06	1.07	1	.32
	"average" vs. "difficult"				
	Intercept	-1.36	.72	1	.06
	proximate switches ratio	2.24	.86	1	.012

which threatens the internal validity of this study. We counteract this learning effect by applying a counterbalance measure design.

Another threat to validity is the choice of JabRef as the subject system. JabRef is written in a single programming language and its code complexity and quality might influence the study. For instance, code with low quality and/or high complexity might result in developers spending more time to read and understand it, and thus longer eye gaze times for certain parts of the code. We tried to mitigate this risk by choosing a generally available system that is an actively used and maintained open source application and that was also used

in other studies. Further studies, however, are needed to examine the effect of factors, such as code quality, to generalize the results.

In our study, JabRef had to be run through the command prompt using ANT and not directly in Eclipse. This meant that participants were not able to use breakpoints and the debugger within Eclipse and might have influenced the results. This restriction might has influenced the way the developers explored the source code and specifically the relatively low call relationships which were observed between the explored methods might be influenced. Further, it is imaginable that generally more lines within a method might be read when using a debugger. We intend to conduct further study to investigate if our findings generalize to other settings, e.g., ones in which the project can be run from within Eclipse.

iTrace collects eye gazes only within Eclipse editors. This means that we do not record eye gaze when the developer is using the command prompt or running JabRef. However, since we were interested in the navigation between the code elements within the IDE, this does not cause any problems for our analysis.

If the user opens the “Find in File" or “Search Window" within Eclipse, or a tooltip pops up when hovering over an element in the code, the eye gaze is not recorded as this overlaps a new window on top of the underlying code editor window and iTrace did not support gazes on search windows at the time of the study. To minimize the time in which eye gazes could not be recorded, we made sure to let participants know that once they were done with the find feature within Eclipse to close these windows so gaze recording can continue.

Finally, most professional developers were mainly Visual Studio users for their work, we conducted our study in Eclipse. However, all professional developers stated that they did not have problems using Eclipse during the study.

4.7 Discussion

Tracing developers’ eyes during their work on change tasks offers a variety of new insights and opportunities to support developers in their work. Especially, the study’s focus on change tasks, the richness of the data, and the finer granu-

larity of the data provide potential for new and improved tool support, such as code summarization approaches or code and artifact recommendations. In the following, we will discuss some of these opportunities.

4.7.1 Richness of Eye-Tracking Data and Gaze Relevance

Our findings show that the eye-tracking data captures substantially more (***O1***) and different aspects (***O2***) of a developer's interaction with the source code. Therefore, eye-tracking data can be used complimentary to user interaction task context to further enhance existing approaches, such as task-focused UIs [Kersten and Murphy, 2006], or models for defect prediction [Lee et al., 2011]. In particular, since eye-tracking data also captures gaze times—how long a developer spends looking at a code element—more accurate models of a code element's relevance could be developed as well as models of how difficult a code element is to comprehend which might inform the necessity of refactoring it.

To examine the potential of the gaze time, we performed a small preliminary experiment to compare a gaze-based relevance model with a model based on user interaction. We focused on professional developers and were able to collect and analyze user ratings from 9 professional developers within the group of participants, also since not everyone was willing to spend additional time to participate in this part. Each developer was asked to rate the relevance of the top 5 elements ranked by gaze time as well as the top 5 ranked by degree-of-interest (DOI) from Mylyn's user interaction context [Kersten and Murphy, 2006] on a five-point Likert scale. Overall, participants rated 76% of the top 5 gaze elements relevant or very relevant and only 65% of the top 5 DOI elements as relevant or very relevant. While these results are preliminary and further studies are needed, the 17% improvement illustrates the potential of the data richness in form of the gaze time.

4.7.2 Finer Granularity of Data and Task Focus

Most current research focuses on how developer build up context on class or method level. Most prominently, editors of common IDEs, such as Visual Studio

or Eclipse, display whole classes, but even the recently suggested new bubble metaphor for IDEs displays full methods [Bragdon et al., 2010]. Similarly, approaches to recommend relevant code elements for a task, such as Mylyn [Kersten and Murphy, 2006, Mylyn, 2015] or wear-based filtering [DeLine et al., 2005b], display the change task context on class and method level. While the method and class level are important, our results show that developers build up their context by focusing only on small fractions (on average 32%) of methods (***O3***). Hence, exploring the fine-grained fragments of a change task context might enable to inform new approaches to identify and highlight the parts which are relevant for the current task.

Since developers focus a lot on data flow within a method (***O4***) that is related to the task, we hypothesize that a task-focused program slicing approach might provide a lot of benefit to developers working on change tasks. Such an approach could take advantage of existing slicing techniques, such as static or dynamic slicing [Weiser, 1981, Korel and Laski, 1988], and identify the relevance of a slice based on its relation to the task by, for instance, using textual similarity between the slice and the task description or previously looked at code elements.

By using eye-tracking to capture a more fine-grained task context while a developer is working, we are also able to better determine what a developer is currently interested in and complement existing approaches to recommend relevant artifacts to the developer, such as Hipikat [Čubranić and Murphy, 2003] or Prompter [Ponzanelli et al., 2014].

Our results also suggest that task contexts can be used to assume a developer's perceived difficulty. Since no change task was successfully solved with a high perceived difficulty, this information could be used to inform approaches which prioritize change tasks to be assigned to developers. Furthermore, if we could recognize when a developer is having difficulty, adequate approaches to help could be provided, such as suggesting expert-developers for that place in the source code.

Furthermore, the fine-grained eye-tracking data also enables to recognize when developers are skimming source code (***O8***). This might inform approaches which depict summaries of source code elements first and if they want to focus

on a specific element the view zooms in and hides remaining irrelevant source code, such that developers are not distracted by proximate source code.

Finally, the insights from our study can also be used to inform summarization techniques to help developers comprehend the relevant parts of the code faster. Existing techniques to summarize code have mainly focused on summarizing whole methods [Haiduc et al., 2010a, Haiduc et al., 2010b] rather than only summarizing the parts relevant for a given task. Similarly, the approach by Rodeghero et al. [Rodeghero et al., 2014] focused on using eye-tracking to summarize whole methods. Our findings show that developers usually do not read or try to comprehend whole methods and rather focus on small method fractions and data flow slices for a change task. This suggests that a more task-focused summarization that first identifies relevant code within a method according to previous eye-tracking data or other slicing techniques and then summarizes these parts of the method, might help to provide more relevant summaries and aid in speeding up code comprehension.

4.7.3 Accuracy of Method Switches

The eye-tracking data captured in our study shows that a lot of the switches between methods are between methods in close proximity, as well as within a class ***O5, O6***. These findings suggest that there is a common assumption among developers that nearby code is closely related. While this is not a new finding, the additional data captured through eye-tracking that is not captured by user interaction monitoring provides further evidence for this switch behavior. This finding also suggests that a fisheye view that zooms in on the current method and provides much detail on methods in close proximity but less on methods further out might support faster code comprehension for developers.

A common assumption of navigation recommendation approaches is that structural relations between elements are important in a developers' navigation [Robillard, 2005]. While empirical studies that examined developers' navigation behavior based on user interactions have shown that developers actually follow such structural relations frequently, in particular call relations (e.g., [Fritz et al., 2014b]), the eye-tracking data of our study shows that developers perform

many more switches that do not follow these relations and that are not captured by explicit user interaction. These findings point to the potential of eye-tracking data for improving method recommendations as well as for identifying the best times for suggesting structural navigation recommendations. However, further studies are needed to examine this possibility.

While developers switch relatively often between methods, they only focus on few methods ***O7***. Exploring further how these methods can automatically be distinguished from the remaining methods, might improve approaches to summarize task contexts which can help resume work faster, be shared with colleagues, or be mined again for further research ideas.

4.7.4 Eye-Tracking for Each Developer

As discussed, using eye-trackers in practice and installing them for each developer not just for study purposes bares a lot of potential to improve tool support, such as better task-focus, recommendations, summarization, or even recognizing the perceived difficulty. With the advances and the price decrease in eye-tracking technology, installing eye-trackers for each developer might soon be reasonable and feasible. At the same time, there are still several challenges and questions to address to be smooth and of value to developers, in particular with respect to eye calibration, granularity level and privacy. Several eye-trackers, especially cheaper ones, currently still need a recalibration every time a developer changes position with respect to the monitor, which is too expensive for practical use. In our study, we recalibrated twice during each session to make sure that we captured eye gazes on the correct source code lines. Further, we also asked the study participants to not make very large head movements (small head movements are natural and taken care of by the eye tracker's headbox). For tool integration, one has to decide on the level of granularity that is best for tracking eye gazes. While more fine-grained data might provide more potential, eye-tracking on a finer granularity level is also more susceptible to noise in the data. Finally, as with any additional data that is being tracked about an individual's behavior, finer granular data also raises more privacy concerns that should be considered before such an approach is being deployed. For instance, the pupil diameter or

the pattern of eye traces might also be used to monitor the cognitive load of the developer, which could also be used in harmful ways.

4.8 Conclusion

To investigate developers' detailed behavior while performing a change task, we conducted a study with 22 developers working on three change tasks of the JabRef open source system. This is the first study that collects simultaneously both eye-tracking and interaction data while developers worked on realistic change tasks. Our analysis of the collected data shows that gaze data contains substantially more data, as well as more fine-grained data, providing evidence that gaze data is in fact different and captures different aspects compared to interaction data. The analysis also shows that developers working on a realistic change task only look at very few lines within a method rather than reading the whole method as was often found in studies on single method tasks. A further investigation of the eye traces of developers within methods showed that developers "chase" variables' flows within methods. When it comes to switches between methods, the eye traces reveal that developers only rarely follow call graph links and mostly only switch to the elements in close proximity of the method within the class. Furthermore, the fine-grained gaze context showed that developers focus only on a few methods when investigating a change task.

These detailed findings provide insights and opportunities for future developer support. For instance, our approach for fine-granular navigation recommendations or our approach to recognize the perceived task difficulty demonstrate the potential of capturing gaze contexts. The findings demonstrate further that method summarization techniques could be improved by applying some program slicing first and focusing on the lines in the method that are relevant to the current task rather than summarizing all lines in the whole method. In addition, the findings suggest that a fisheye view of code zooming in on methods in close proximity and blurring out others, might have potential to focus developers' attention on the relevant parts and possibly speed up code comprehension.

The approach that we developed for this study automatically links eye gazes to source code entities in the IDE and overcomes limitations of previous studies by supporting developers in their usual scrolling and switching behavior within the IDE. This approach opens up new opportunities for conducting more realistic studies and gathering rich data while reducing the cost for these studies. At the same time, the approach opens up opportunities for directly supporting developers in their work, for instance, through a new measure of relevance using gaze data. However, possible performance and especially privacy concerns have to be examined beforehand.

Acknowledgement

The authors would like to thank the participants in the study. The authors would also like to thank Meghan Allen for her helpful feedback. This work was funded in part by the SNF grant DARINO (200021_ 150050) and an ABB grant.

5

Using Eye Gaze Data to Recognize Task-Relevant Source Code Better and More Fine-Grained

Katja Kevic

Published at the 39th International Conference on Software Engineering Companion, 2017

Contribution: Study design, data collection, data analysis, and paper writing

Abstract

Models to assess a source code element's relevancy for a given change task are the basis of many software engineering tools, such as recommender systems, for code comprehension. To improve such relevancy models and to aid developers in

finding relevant parts in the source code faster, we studied developer's fine-grained navigation patterns with eye tracking technology. By combining the captured eye gaze data with interaction data of 12 developers working on a change task, we were able to identify relevant methods with high accuracy and improve precision and recall compared to the widely used click frequency technique by 77% and 24% respectively. Furthermore, we were able to show that the captured gaze data enables to retrace which source code lines developers found relevant. Our results thus provide evidence that eye gaze data can be used to improve existing models in terms of accuracy and granularity.

5.1 Research Problem and Motivation

Software developers working on change tasks spend a majority of their time navigating and reading source code [Ko et al., 2006]. A variety of software development tools emerged to support developers during this code comprehension phase. These tools range from recommender systems to identify code elements that should be visited next [Robillard, 2005, Čubranić and Murphy, 2003] or web documents that might be interesting [Ponzanelli et al., 2014, Čubranić and Murphy, 2003], to tools that keep track of which source code elements have already been visited and assessed to be relevant [Kersten and Murphy, 2005, Kersten and Murphy, 2006, Bragdon et al., 2010], to approaches which create summaries of the source code [Moreno et al., 2013, Panichella et al., 2016]. Underlying all of these approaches is a model that determines the current relevancy of a source code element (e.g., class, method, or even statements). A more accurate relevancy model thus bears great potential for a large variety of tool support for code comprehension.

Existing relevancy assessment models mainly differ in the data source which is analyzed to infer recommendations from, ranging from structural code analysis [Robillard, 2005], to the mining of commit histories [Ying et al., 2004, Zimmermann et al., 2004], to the analysis of clicks made in the code for the task [Kersten and Murphy, 2005]. The relevancy models based on these data sources are generally limited by the granularity and the detail captured in the data source.

For example, in the case of click analysis only the elements which developers clicked on are considered while the elements which were only read are omitted. Recent advances in eye tracking technology however enable to capture a more complete and fine-grained picture of the developers' activities during source code comprehension.

In our first exploratory study in which we used eye tracking technology with 12 professional and 10 student developers [Kevic et al., 2015, Kevic et al., 2017], we found that developers look at substantially more code elements than they click on and that the eye gaze data captures different aspects about the developers' way of work than the click based data. We further found that developers read only few lines (on average 32%) of source code methods. Hence, it is important to investigate whether relevancy models can be applied to a finer code granularity, such as single lines of code. Building on this work, we conducted a second exploratory study to investigate whether click based data and gaze based data can be used complementary to improve relevancy models for code elements. The better determination of the relevant elements within the navigation steps, can improve tools which help developers keep track of their work or can be used to resume not yet finished tasks. Furthermore, with a better knowledge about the relevant source code elements, recommendations for other source code artifacts might be improved as well.

5.2 Related Work

Most related to this work are approaches which assess a code element's relevancy and approaches which use eye trackers to better understand how developers read source code.

5.2.1 Approaches to Determine a Code Element's Relevancy

The data sources which have been used to determine the relevancy of a code element for a given task can be categorized into historical and source code related data sources. Historical data sources stem from tracking developers'

activities in the source code. For example, commit histories (e.g., [Ying et al., 2004, Zimmermann et al., 2004]) or interaction logs (e.g., [Kersten and Murphy, 2006, DeLine et al., 2005a, Parnin and Gorg, 2006]) have been analyzed to infer the relevant source code elements. Source code related data sources use structural (e.g., [Robillard, 2005]) or textual (e.g., [Čubranić and Murphy, 2003]) relations between source code elements, or combine several of these data sources (e.g., [Piorkowski et al., 2012]) to determine a source code's relevancy. While many different data sources have been investigated to assess a code element's relevancy, none of the approaches used the more fine-grained eye gaze data to assess the relevancy of explored methods for developers working on change tasks.

5.2.2 Eye Tracking in Software Engineering

Recent advances in technology, however, allow us to use eye trackers to gain a deeper understanding about developers' cognitive processes when navigating and reading source code. One of the first studies conducted by Crosby and Stelovsky [Crosby and Stelovsky, 1990] compared the way developers read an English and a Pascal version of an algorithm. Others studied how identifier styles influence developers' eye gazes [Sharif and Maletic, 2010a, Binkley et al., 2013] or how UML diagrams are being read [Yusuf et al., 2007, Sharif and Maletic, 2010b]. Most of these studies focus on small code snippets, while we analyze developers' eye gazes for real change tasks of open-source systems.

5.3 Second Exploratory Study and Uniqueness

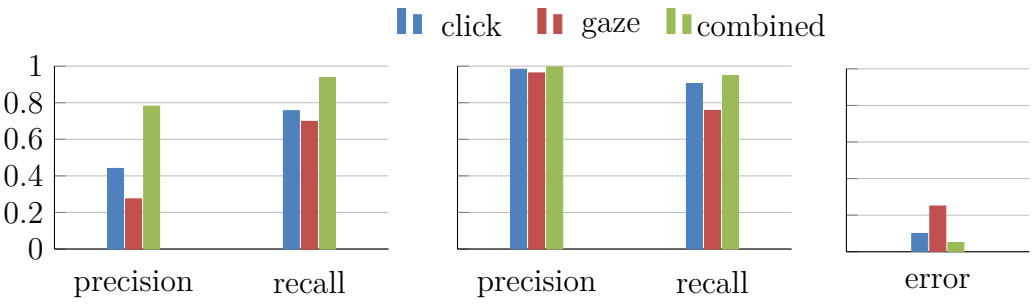
The aim and uniqueness of this research is to discover if on-line eye tracking is beneficial for improving code relevancy models and for supporting developers working on a change task. To this end, we analyzed the data gathered in a second exploratory study with 12 students from the University of Zurich[1].

In this study, we asked the participants to work on real change tasks of the open source system Gson [Gson, 2016]. After the participants worked for ten

[1]This study is part of a paper that will be under submission soon

minutes on the first change task, the experimenter interrupted the participants and showed them a list of the source code elements they interacted with. We asked them to point out the relevant ones. After the participants pointed out the relevant source code elements, they continued to work on the task. The experimenter interrupted the participants a second time after ten minutes and followed the same procedure. We further asked the participants if they have time to work on another change task and seven of the participants agreed. For this change task, the experimenter waited until the participants worked for five minutes on the task and then took a note of the method they were looking at the moment. The experimenter then waited until the participants noticeably navigated to another method (i.e., through the use of a navigation support tool) and interrupted the participants. The experimenter asked the participants to go back to the previous method and point out the lines in the method which are relevant. This procedure was repeated twice.

Throughout the study period, we used an Eye Tribe ET1000 eye tracker and the iTrace plugin [Shaffer et al., 2015] to log participants' eye gazes, in particular the classes, methods and lines of code the participants looked at. In addition, we used a self-developed IDE plug-in to capture the classes and methods that participants clicked on in their IDE. We collected in the first part of the study relevancy assessments for 181 methods, out of which 45.9% were marked as relevant. In the second part, we collected for 14 methods fine-grained relevancy assessments.



(a) Relevant methods. (b) Irrelevant methods (c) Error rate

Figure 5.1: Precision, recall and error rate of methods' relevancies.

5.4 Results

In section 5.4.1 we analyzed the data from the second part of the study and in section 5.4.2 we analyzed the data from the first part of the study.

5.4.1 Relevancy within Source Code Methods

We found that developers gaze significantly longer ($p = .027$) and more often ($p = .022$) at the lines they assessed to be relevant (Mean (M) duration = $0.8s$, M gaze frequency = 64.3) than at lines they did not assess to be relevant (M duration = $0.2s$, M gaze frequency = 16.8). Further, they also fixated more often the lines they assessed to be relevant ($p = .022, M = 44.1$) than the lines they did not assess to be relevant ($M = 7.9$). Our analysis further revealed that the participants overall only identified few lines within methods relevant. On average, over all participants, only 11% (Standard Deviation (SD) = 8%) of the source code lines within methods were identified to be relevant.

5.4.2 Relevancy of Source Code Methods

As biometric sensors capture individual differences in a more distinctive way [Müller and Fritz, 2016], we ran a binomial regression on the methods in the captured log files for each participant to predict the relevant, respectively irrelevant methods automatically. For the click-based data we considered how often a developer clicked on a specific method [Parnin and Gorg, 2006] and for the gaze-based data we considered how often a developer looked at a specific method. We calculated the precision, recall, and error rate for the methods classified as relevant and for the methods classified as irrelevant (see Figure 5.1). We found that the combination of click-based and gaze-based data recognizes the relevant methods better than using only one source of information. In particular, the precision to identify relevant methods is increased by 77% compared to using only click-based data and by 185% compared to using only gaze-based data. The recall to identify the relevant methods is also increased by 24% compared to click-based data and 34% compared to gaze-based data.

Our analyses of the gathered eye gazes show that on-line eye tracking is beneficial for developers. First, the relevant source code lines within methods can be recognized. Second, the recognition of the relevant source code methods can be improved through combining click-based and gaze-based information. These results are encouraging, since we used a comparatively low-cost eye tracker in this study.

5.5 Contribution

Our analyses revealed that developers find only few lines within methods relevant (11%) and that on-line eye tracking can be used to identify them. We further showed that the combination of click-based and gaze-based information bears potential to improve the relevancy measures for source code elements. Being able to improve a source code element's relevancy measure has the potential to improve a great variety software development tools.

6

Characterizing Experimentation in Continuous Deployment: a Case Study on Bing

Katja Kevic, Brendan Murphy, Laurie Williams and Jennifer Beckmann
Published at the 39th International Conference on Software Engineering, Software Engineering in Practice, 2017
Contribution: Data processing, data analysis, and paper writing

Abstract

The practice of continuous deployment enables product teams to release content to end users within hours or days, rather than months or years. These faster deployment cycles, along with rich product instrumentation, allows product

teams to capture and analyze feature usage measurements. Product teams define a hypothesis and a set of metrics to assess how a code or feature change will impact the user. Supported by a framework, a team can deploy that change to subsets of users, enabling randomized controlled experiments. Based on the impact of the change, the product team may decide to modify the change, to deploy the change to all users, or to abandon the change. This experimentation process enables product teams to only deploy the changes that positively impact the user experience.

The goal of this research is to aid product teams to improve their deployment process through providing an empirical characterization of an experimentation process when applied to a large-scale and mature service. Through an analysis of 21,220 experiments applied in Bing since 2014, we observed the complexity of the experimental process and characterized the full deployment cycle (from code change to deployment to all users). The analysis identified that the experimentation process takes an average of 42 days, including multiple iterations of one or two week experiment runs. Such iterations typically indicate that problems were found that could have hurt the users or business if the feature was just launched, hence the experiment provided real value to the organization.

Further, we discovered that code changes for experiments are four times larger than other code changes. We identify that the code associated with 33.4% of the experiments is eventually shipped to all users. These fully-deployed code changes are significantly larger than the code changes for the other experiments, in terms of files (35.7%), changesets (80.4%) and contributors (20.0%).

6.1 Introduction

As the software industry has moved towards a service model, different companies have adopted techniques such as continuous deployment, in which software is continually released to users [Olsson et al., 2012]. Increasing the rate of releasing software, radically changes the way software is developed and deployed [Bosch, 2012]. Previously product changes occurred as part of major releases while in continuous deployment products evolve. Some organizations have chosen to

couple this rapid deployment with an experimental framework to assess the impact of changes on the end user using a practice referred to as continuous experimentation [Fagerholm et al., 2014]. Some products, such as Bing, have been using online controlled experiments, since 2009 [Kohavi et al., 2009a]. Visibility of continuous experimentation increased with the build-measure-learn cycles advocated in the Lean StartUp methodology [Ries, 2011] in 2011 based upon experiences at IMVU. As the value of continuous experimentation is more and more recognized, large organizations, such as Facebook, Google, and Netflix, increasingly employ continuous experimentation [Parnin et al., 2017].

These incremental, rapid deployments offer the opportunity for development teams to formulate hypotheses about expected user behavior due to a software change, define metrics needed to be collected to verify the hypotheses, and continuously learn how users react. The process to verify hypotheses is through controlled experiments[1] [Mason et al., 1989]. Different versions of the product are exposed to randomly-chosen user subgroups. By measuring users' behavior in each group, development teams have the ability to make a data-driven decision of whether to modify, delay, or abandon the given software change [Kohavi et al., 2009a, Kohavi et al., 2009b, Lindgren and Münch, 2015]. If a software change is abandoned the code associated with it, is removed from the system.

While a few works investigated the experimentation process, they often focus on the use of the process to evolve small products or services (e.g., [Lindgren and Münch, 2015]), or share experience reports and lessons learned (e.g. [Kohavi et al., 2013, Tang et al., 2010]). The full life-cycle from an experiment's first code change all the way to the analysis of the captured usage measurements has not been characterized. Parts of product strategies evolve based on experiments' outcome [Fagerholm et al., 2014, Fagerholm et al., 2017]. Therefore, knowing how quickly a product team can learn from experiments may help to better plan product strategies. Furthermore, knowing more about the code changes used for experiments may allow the elaboration of different experimentation procedures tailored to different kinds of code changes. Finally, we determine how

[1] Also called A/B tests, split tests, bucket testing, randomized experiments, online field experiments, canary, flighting, or gradual rollouts.

many experiments are ultimately deployed to all users. Knowing more about the amount of deployed experiments helps to assess the efficiency of continuous experimentation approaches for products in different maturity stages. Previous research has not addressed how the code changes for experiments that were deployed to all users differ from the code changes which were not deployed to all users. Knowing more about these differences might enable efficiencies in the product development and experimentation processes.

The goal of this research is to aid product teams to improve their deployment process through providing an empirical characterization of an experimentation process when applied to a large-scale and mature service. In particular, we investigate the following research questions:

> **RQ1**: What are the characteristics of experiments and their development efforts, in terms of time spans, number of people involved, files and changes in a large-scale and mature product?
>
> **RQ2**: What percentage of experiments are ultimately deployed to all users?
>
> **RQ3**: How do the experiments which are deployed to all users differ from the experiments which were not deployed to all users in terms of time spans, number of people involved, files and changes?

To answer these questions, we conducted a large-scale empirical analysis of Bing, Microsoft's search engine. We analyzed 21,220 experiments conducted in Bing since 2014, and all code changes that occurred during the same period of time. These experiments include a variety of different kinds of hypotheses that are tested. These hypotheses range from testing tweaks in algorithms to the impact of user interface or configuration changes on the end users. Through establishing a procedure to link specific change sets within Bing's change history to specific experiments, we analyzed the code changes committed for experiments. A change set includes one or multiple files changed at the same time. Through this analysis, we inferred whether the code changes for an experiment were ultimately

deployed to all users. Change sets which we could not link to experiments were also analyzed.

The remainder of this paper is structured as follows. First, we present background on continuous experimentation and the related work which has been conducted in this area. Then, we describe how experiments are conducted within a large-scale and mature product, i.e. Bing. We describe the main points which increase the complexity of the experimentation process. Section 6.4 describes the historical data that we used to analyze characteristics of experiments and infer whether the software change for the experiment was ultimately shipped to all users. The results of this analysis are described in Sections 6.5,6.6, and 6.7. We then discuss the threats to validity in Section 6.8, our findings in Section 6.9, and conclude our work in Section 6.10.

6.2 Background and Related Work

In this section, we provide background and related work on continuous deployment and continuous experimentation.

6.2.1 Background

We define and differentiate four terms used in this paper:

Continuous Integration. Software is developed in smaller, incremental change sets which are regularly integrated into the codebase of the complete product, where a process automatically builds and runs a test suite daily, hourly, or even per individual change [Duvall et al., 2007].

Continuous Delivery. The automated implementation of an application's build, deploy, test, and release process [Humble and Farley, 2010].

Continuous Deployment. A continuation of the continuous delivery process, where the application or service is automatically deployed to the customer [Humble and Farley, 2010].

Continuous Experimentation. All changes require a clear hypothesis of their impact on the end customer, and that hypotheses are verified against a subset of customers prior to full deployment [Fagerholm et al., 2014].

We found that the terms delivery and deployment are often incorrectly used interchangeably in literature on this subject.

One of the prerequisites for continuous experimentation, is that a product team deploys code changes frequently through continuous delivery or continuous deployment. Continuous integration enables both, continuous delivery and continuous deployment processes.

Through verifying the product at both, unit and system level, bugs can be detected soon after they have been introduced, and the quality of the software can be measured and analyzed over time. For example, the Apollo space mission, in the 1960s, incorporated all changes made during the day into a single overnight computer run [Mindell, 2008]. Hence, developers can be increasingly confident about the quality of their code change. Further, by including feedback mechanisms into each step in the continuous integration pipeline, developers have the possibility to react immediately to merge conflicts, to bugs or to irregularities within the collected measures. One of the main benefits of employing the principles of continuous integration is that the product remains in a deployable state and could be released at any point in time.

Further advancements in technology beyond continuous integration enabled continuous delivery and continuous deployment practices. These later two practices originated in the Software-As-A-Service area, whereby changes to the code base could be rapidly deployed to the service and the impact of these changes on the end users can be measured.

In an experiment, different versions of the product are exposed to different randomly chosen user groups. One version of the product includes a change or a new feature, referred to as the treatment, and the other version is the current version of the product, referred to as the control [Mason et al., 1989].

For each experiment a prior hypothesis is formulated which states that the treatment is not better than the control when evaluated with a measure[2], which measures the targeted aspect of the user behavior [Kohavi et al., 2009b]. As the experiment runs for a predefined amount of time, the initial hypothesis is evaluated through testing for statistical differences between the treatment and the control. If the null hypothesis can be rejected, the users, in fact, react differently to each version of the product.

Five main components enable developers to run experiments [Kohavi et al., 2009b, Fagerholm et al., 2014, Fagerholm et al., 2017, Olsson et al., 2012]:

1. a hypothesis on the experiment's objective which is modeled in measurable metrics;
2. the instrumentation of the product;
3. a randomization algorithm;
4. an assignment method;
5. and a data path.

The product is instrumented such that the metrics defined to verify the hypothesis can be captured. The randomization algorithm is used to identify the users that are exposed to either the treatment or the control of an experiment. One difficulty for a large-scale product in which parallel experiments are run, is that the randomization algorithm has to ensure that there are no correlations between the assignments of experiments. The assignment method is the mechanism in place used to route user requests to the specified version of the product.

Users can be assigned to specific version of the product using techniques, such as traffic splitting, page rewriting, client-side assignment, and server-side assignment. Kohavi et al. [Kohavi et al., 2009b] elaborate the advantages and disadvantages of each method. Finally, the data path is responsible for collecting the defined metrics and preparing the statistical analysis.

[2]Also called the overall evaluation criterion (OEC), response, dependent variable, outcome, evaluation metric, key performance indicator, endpoint or fitness function.

6.2.2 Continuous Experimentation at Microsoft

Different works analyzed the experimentation process within Bing. Kohavi and colleagues [Kohavi et al., 2013, Kohavi et al., 2014, Kohavi and Longbotham, 2016] and Crook and colleagues [Crook et al., 2009] share their insights and lessons learned while running an experimentation process at a large-scale. They work out seven rules of thumb for running controlled experiments and seven pitfalls to be avoided when running controlled experiments. They identify three main categories of challenges, including organizational challenges, engineering challenges, and the challenge of having a trustworthy experiment outcome. While Kohavi et al. [Kohavi et al., 2009a] further look into the cultural aspects and share valuable real-world examples, Kohavi et. al [Kohavi et al., 2009b] focus on the technical aspects in more detail and summarize the cost of experimentation when using different assignment methods. The trustworthiness of experiments is further elaborated through the analysis of five experiments' outcomes by Kohavi et al. [Kohavi et al., 2012]. Deng et al. [Deng et al., 2013] investigate how the percentage of users to which the experiment is exposed or the exposure duration of the experiment can be reduced while the same statistical power can be observed. Deng [Deng, 2015] explores an objective Bayesian A/B testing framework to analyze metrics. In this paper we build upon that work with a focus on the full life-cycle of experiments, characterizing experiment and code changes attributes.

6.2.3 Other Continuous Experimentation Research

Several case studies have been conducted which identified the challenges which are faced when employing experimentation. Other researchers [Davenport, 2009, Kohavi et al., 2009a, Kohavi et al., 2013, Lindgren and Münch, 2015] have identified the cultural shifts often necessary in development teams to be one of the major challenges. In particular, the risk of individuals losing power or prestige due to experiment results contrary to their own intuitions and the importance of a consistent reward system which rewards the volume of valuable experiments regardless of outcome have been observed as the main cultural

challenges. Lindgren and Münch [Lindgren and Münch, 2015] further identified that slow development cycles, the product instrumentation and the identification of the metrics to measure the user experience are further challenges. Rissanen and Münch [Rissanen and Münch, 2015] largely confirmed these challenges when they studied experimentation in a B2B environment. They further found that the capturing and transferring of user data becomes a further challenge, as legal agreements come into play.

Fagerholm et al. [Fagerholm et al., 2017, Fagerholm et al., 2014] explore a model of continuous experimentation and how experiments are related to the vision and the strategy of a startup company's product. They found that the results from experiments altered the strategy of products, but the vision of the product remained unchanged. Within their suggested model, called RIGHT, the experimentation process is structured into build-measure-learn blocks. In our research, we approximate the duration of such a block.

While these case studies and experience reports focused on identifying challenges within an experimentation process and analyzed how experiments influence a product's strategy, we focus on the source code development efforts which are involved in an experimentation process.

6.2.4 Experimentation - the State of Practice.

Systematic experimentation processes are prevalent in large companies that offer SaaS services [Bakshy et al., 2014, Bosch, 2012, Kohavi et al., 2013, Lindgren and Münch, 2015, Xu et al., 2015]. For example, at Google every change that can impact customers goes through an experimentation process [Tang et al., 2010]. Thereby, many types of changes to the product are run as experiments: from visual enhancements to changes within back-end algorithms. These companies have developed scalable platforms which offer the infrastructure to run experiments in a systematic way. Many of these advanced experimentation platforms have further tools to support the data analysis integrated. For example, LinkedIn's XLNT analysis dashboard [Xu et al., 2015] supports experimenters to make a data-driven decision of whether the experiment improved the user experience by presenting

summarized views. Other tools and platforms to run systematic experiments are emerging. Google's Analytics experiment framework [GoogleAnalytics, 2016] and Facebook's PlanOut [PlanOut, 2016] are two examples of such frameworks that support an experimentation process.

When Lindgren and Münch [Lindgren and Münch, 2015] surveyed ten smaller software companies to understand the current state of the practice of experimentation processes applied, they found that the surveyed companies recognize the value of experimentation but only few companies run systematic experiments often. As more and more services and even desktop applications such as Chrome or Mozilla Firefox, adapt principles of continuous delivery [Adams and McIntosh, 2016], experimentation can become an integral part within the development cycle of a wide range of different products.

While all these case studies and experience reports enable important insights into different experimentation processes, we add to the existing body of research the first empirical study on a large-scale and mature experimentation process. In particular, compared to previous works, we describe the full life-cycle of an experiment from the first code change to the deployment of the experiment.

6.3 Bing Experimentation Process

For this case study, we analyze Microsoft's search engine Bing. Bing includes the main search results pages from Bing.com, as well as several services that are consumed by other Microsoft products, such as Cortana. Bing's richness in a variety of services enabled us to study the experimentation process in different environments. While the majority of the services are customer based, some are development support services for the rest of Bing (e.g. developing the deployment software). Bing is broken down into a large number of independent components, where components are either library components or dedicated to specific services. Since 2009, Bing and other services across Microsoft, increasingly use the Experimentation Platform (ExP). ExP was introduced by the Experimentation Platform team within Microsoft that was formed in 2006. ExP is a highly scalable platform that enables a systematic experimentation process [Kohavi et al., 2009a].

In the following, we characterize the individual steps of in the experimentation process in Bing (see Figure 6.1).

6.3.1 Experiment Design

In a first step, developers formulate a hypothesis that defines the aspects of the users' behaviors they seek to improve. Then, they identify the set of metrics that allow the formulated hypothesis to be tested. ExP provides a wide range of predefined metrics that can be used to capture the users' behaviors. If this set of predefined metrics does not properly test the developers' hypothesis, the developers need first to implement or request the needed instrumentation within the product to capture additional aspects of the users' behaviors. The set of metrics that is identified for the experiment are then captured within an ExP scorecard. Furthermore, experimenters need to decide on the number of users that are exposed to each group within the experiment and the amount of time the experiment will be exposed to the users. A rigorous experiment design is indispensable for being able to make a data-driven decision of whether the feature should be deployed.

6.3.2 Pre-Study

Development teams have the possibility to rapidly evaluate a predetermined hypothesis by creating an internal pre-experiment prior to fully developing the software change. Internal experiments are usually mock-ups or quick-hacks of the idea that are submitted to an internal crowd-platform. Within this crowd-platform, the mock-ups or quick-hacks are shown to a chosen set of people, without identifying which is the treatment and which is the control. The outcome of these human judgments is then used to evaluate if the idea should be further implemented and then run through the full experimentation process or if the idea does not show potential. Furthermore, product teams use the outcomes of these internal experiments to prioritize the planned experiments.

6.3.3 Source Code Development and Deployment

The development team for each of the Bing services has the autonomy to choose their own software development process. Each service has its own development environment managed through its own branching structure. Also the deployment process varies among the different services and is often based on the characteristics of the service itself. For instance, the service that manages the user interface (UI) has an hourly development and deployment cycle, where the deployment process rolls out the changes in a controlled manner and rolls back changes that have bugs. Conversely, the development and deployment of complex state based services, such as the index server can require additional verification: deployment cycles can be weekly or longer.

6.3.4 Experiment Execution

After the source code is changed and deployed, the experiment execution starts. Experiments generally run for one or two weeks. To lower unforeseeable risks of system failures, an experiment generally starts by directing a small percentage of users to the advanced version, the treatment, of the product. After some time, where no failures are detected, the percentage of users directed to the treatment gradually increases. This mechanism ensures that if there was an issue with an experiment only a small percentage of users experienced it. There are different metrics which are continuously captured while the experiment is running. One group of metrics, the *guardrail* metrics, is the sentinel to the health of the product. If metrics in this group change drastically, egregious issues with the experiment are detected and ExP informs an alert system, which shuts the experiment automatically down and all traffic will be sent to the prior version. An example of a guardrail metrics is the page load time.

Since ExP allows multiple experiments to run in parallel, the risk of different experiments interacting with each other increases. Because the interaction of experiments can corrupt the metrics captured for each experiment and possibly harm the user experience with the product, it is pivotal that a possible interaction is prevented. If the prevention was bypassed, the corruption it is quickly detected.

ExP incorporates mechanisms to prevent and detect interactions. To prevent interactions between different experiments, each experiment defines constraints. These constraints are used to identify the experiments that should not be exposed to the same user. To detect interactions between running experiments, ExP scans and analyzes the metrics of pairs of running experiments. If interactions between experiments are recognized, an alert is raised and the owners of the experiment involved in the interaction are informed. They then decide whether to stop one of the experiments.

If a bug in the changed code or in the experiment configuration is detected, another alert is raised which informs the owners of the experiments. If no issues are detected during the experiment execution, the experiment is stopped automatically after the exposure duration specified by the experimenter.

6.3.5 Data Analysis

To inform experimenters of the status of a running experiment, ExP allows the creation of scorecards on a periodic basis. A more extensive data analysis occurs after the experiment completed.

If the experiment ran without any issues (i.e. no alerts from the alert system reported and no bugs in the source code detected), it is considered to be a valid execution of the experiment. Developers can now decide between three alternatives: deploy the experiment, abandon the experiment, or iterate the experiment. Each experiment within Bing aims to improve user behavior on two levels. The first level is the same for each experiment within Bing. Metrics in this level are called the *main* metrics which each experiment tries to improve. These metrics target long-term goals of the product, such as the number of clicked search results. The second level is experiment-specific and targets the metrics that were defined in the product team's hypothesis for the experiment. Examples of experiment-specific metrics are the elapsed time until a first search result is clicked or whether a suggested query completion was used. Based on the product team's hypothesis and the gathered metrics a data-driven decision of whether to ship, abandon, or iterate the code change is made.

If the overall evaluation criterion (OEC) measurably improved with the new version of the product, then the treatment of the experiment is shipped and abandoned in the contrary. In practice, the OEC takes several factors into account, such as the user experience and revenue, and allows to trade one factor off for another. If the product team cannot make a data-driven decision based on the metrics that were collected, the product team iterates on the experiment design and defines a new set of metrics to test the hypothesis on. If the correct metrics have been collected for the experiment, but more user data is needed to enable a data-driven decision, then a new iteration of the experiment is launched. Finally, the product team can also decide to iterate on the source code change, but to validate the same hypothesis.

If the experiment executed with issues, then it is an invalid execution. If there was a bug in the changed code or in the experiment configuration detected, the product team iterates on a further code change to eliminate the bug. If the experiment was stopped because the metrics indicated that they were harming the user experience, then it might be abandoned.

6.3.6 Complexity of Experimentation

We observed that running a thorough experimentation process on a large scale service is very complex. Figure 6.1 depicts the experimentation process currently used by Bing. The complexity of the experimentation process stems from different aspects. First, while it is possible to rapidly verify that a deployment does not break the user experience, it takes time to verify that the user experience is improved or not degraded by the change. Experiments have to be exposed to the user groups for at least one week. There are many reasons for this: one reason is that users interact with the product differently on different days of the week. While it is imaginable that users search, for example, may be more work related on a Monday morning, they would rather search for social related activities over the weekend. Another reason is that it is important to have enough users for statistical validity for trustworthy comparison. Depending on the size of the change and the prominence of the feature, it takes time until a large enough number of users interacted with it to gain enough data for statistical

validity. Second, since Bing has users all over the world and runs on different devices, the results of the experiment can be country and device dependent. This segmentation adds complexity to the configuration of the service. Third, there is a limited capacity to run experiments. Parallel experiments can be run for each deployment and each data center. Finally, after running a series of experiments it needs to be tested how these experiments interact with each other and whether the combined user experience is still improved.

6.4 Study Data and Method

Our dataset consists of 21,220 experiments that were conducted within Bing over the last 2.5 years. Bing offers the possibility to study experimentation in inherently different components of the product. Since these different components have slightly different procedures to capture and implement experiments, the following analysis does not capture all experiments for all components within Bing.

6.4.1 Experiments

As the experimentation process within Bing emerged and changed over time, we restricted the analysis to only experiments that were created since the beginning of 2014. Our dataset comprises historical data of 21,220 experiments run within 19 components of Bing. We downloaded information about these experiments through an API offered by ExP. ExP stores attributes about experiments and stores the exposure duration of each experiment, which reflects the amount of time the experiment's treatment was exposed to end users of the product. In our analysis, we included only experiments for which a positive exposure duration was stored (occasionally experiments were created, but never run and hence the exposure duration is zero). Furthermore, ExP stores for each experiment a list of people who are responsible for the experiment, i.e. the *owners* of the experiments. Finally, we also retrieved the information showing which

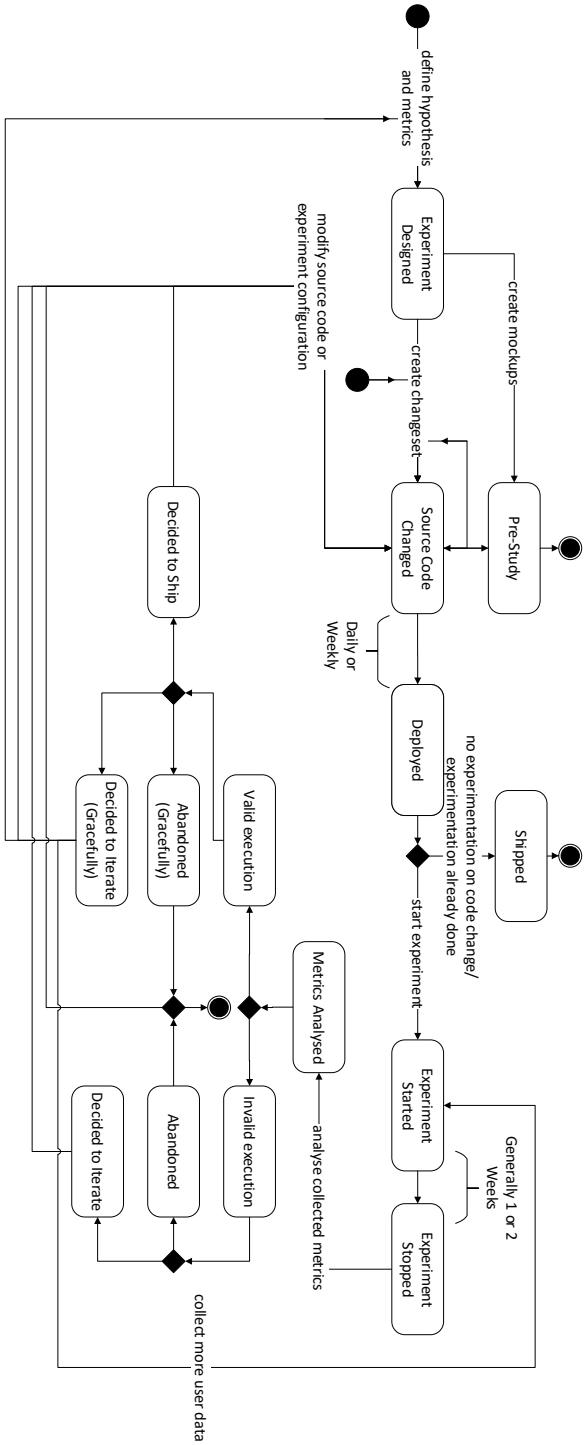


Figure 6.1: Experimentation process used in Bing.

experiments are iterations of one another (i.e. the experiments which belong to the same experiment group).

Using another API offered by ExP, we downloaded for each experiment the created scorecards, which include several metrics measured over the experiment duration. ExP generates scorecards on a regular basis throughout an experiment. Hourly and daily scorecards measure the early hours of experiments and look for serious negative results that indicate a regression or bug in the product. As time goes on, the system generates fewer scorecards, because the bugs are typically detected early on, and the goal is now to determine the validity of the hypothesis. In our analysis, we considered only the last scorecard that was created for a particular experiment.

6.4.2 Linking Source Code and Experiments

No explicit link exists between experiments and the source code. To re-establish this link, we analyzed all change sets within Bing's source code change history since 2014.

Experiments in Bing are generally controlled through configuration initialization files (INI). As Bing is a large product, consisting of multiple components and developed by hundreds of developers, different syntax are used to configure experiments.

In a first step, we filtered all change sets identifying those that include the editing of at least one INI file. In a second step, we iterated through all the INI files identified in the first step and parsed the files using a regular expression to identify those INI files used to configure experiments. In a third step, we iterated through all INI files identified in the second step. We parsed each version of the files using another regular expression to identify whether the specific file version includes a configuration for one of the 21,220 experiments in our data set. Through this method, we created a link between a specific change set and a specific experiment.

As a result of this analysis, we were able to categorize and label every change occurring in the Bing development environment since 2014 into one of the following four categories:

Matched Change. The change set includes an INI file that was linked to a particular experiment.

High Probability Change. The change set includes an INI file for which we know at least one version has been used to configure experiments.

Low Probability Change. The change set includes at least one arbitrary INI file, but the INI file contains no syntax that implies it is used to configure experiments.

Other Code Change. The change set includes no INI file.

Due to the variety of complex syntax used in INI files, we concede that we may missed matched changes. However, these experiments are represented in the high probability category.

Prior to releasing the code for an experiment, a development team may iterate the code multiple times. Each code iteration is referred to as a change set. The change sets prior to the deployment of the code for an experiment are referred to as *related change sets*. To identify how much effort goes into an experiment, related change sets must be identified.

To identify the related change sets, we use the fact that a file in Bing has 1.3 iterations per year. As a result, we made the assumption that if the same file changes within a 5 day period then we can assume that the change sets containing the file are related. The small percentage (0.07%) of files that change very frequently (greater than 15 times per year) are excluded from the analysis.

The following algorithm is applied to identify and process all related changes. Every change set edited by Bing since 2014 is processed, starting with the latest change set and working backwards. For each file in the change set the process identifies if the file was previously edited within the 5 day time window. If so, the change label for the change set, that the file belongs to, is altered based on the value of the label of the initial change set. If the label on the initial change set is matched change it overrides all other categories. If the label is high probability this overrides low probability and other code changes and if it was low probability this overrides other code changes. Walking backwards through the change sets will result in a cascading effect, where edits that occur within 5 days of the re-labeled change sets will also be re-labeled.

We also analyzed the identification of related changes over a time span of ten days. As the association of related changes remains roughly the same, we decided to use a time span of 5 days in this analysis.

6.4.3 Parsing the Experiment Outcome

The experiments for which we could identify one or more matched changes, allowed us to infer whether the treatment of the experiment was ultimately deployed to all users. In particular, we analyzed the sequences of changed lines (diffs) within the matched changes of an experiment.

Occasionally, the configuration names of experiments are reused. In this circumstance, we cannot infer whether the treatment of the experiment was shipped or not. The syntax for controlling the shipment of experiments' treatments is complex. The parser currently does not cover all options.

6.5 Experiment Characterization (RQ1)

> **RQ1**: What are the characteristics of experiments and their development efforts, in terms of time spans, number of people involved, files and changes in a large-scale and mature product?

We answer this research question from two perspectives. First, we analyze how much time an average experiment within Bing takes (Section 6.5.1). Further, we characterize other attributes of the development efforts and of experiments, such as the people who are involved. Due to substantial differences between components of Bing, we summarize these features for each component separately in Table 6.1. Second, we analyze Bing's change history of the past 2.5 years and compare the changes that are used for experiments to those changes not used for experiments (Section 6.5.2).

6.5.1 Experiment Life-Cycle

Change sets for experiments are rapidly deployed. The average time between the first code change for an experiment and its deployment (last code change observed

before the start of the experiment) is 1.5 days ($SD = 1.4$). Depending upon the specific component of Bing, an experiment iteration is generally exposed for one or two weeks to a user group. On average, experiments are iterated 1.8 times ($SD = 1.8$) and owned by 4.8 ($SD = 2.3$) people. On average, 1409 different metrics ($SD = 488$) are collected for an experiment. Over an experiment group, we observed 6.4 separate changes to software files submitted by 2.3 people ($SD = 1.7$). See Table 6.1 for details on the major Bing components. The analysis of the captured data and additional code changes between iterations adds additional time to the execution of the experiments. Our analysis identified that the experimentation process, from the start of the experiment to the completion of the last iteration of an experiment, takes an average of 42 days, including multiple iterations of one or two week experiment runs. Through characterizing the life-cycle of experiments, a product team is enabled to identify potential bottlenecks. Knowing where the bottlenecks are within the development cycle, enables to appoint either more resources or synchronize resources in an improved way.

6.5.2 Experimental Activity within Bing

As described in Section 6.4.2, we categorized each code change in Bing's change history into one of the four categories: matched change, high probability change, low probability change, and other code change. We assume that many of the changes that we could not link to experimental activity and hence were grouped into the other code changes category are tool-based changes, test related, or bug fixes. Of the changes that we could link to experimental activity, we grouped 12.1% into the matched category, 45.5% into the high probability category and 42.4% into the low probability category. We observed that changes that are related to the matched and high probability category include more files than changes which are categorized into the low probability or other change category. We observed, on average, 78.9 files ($SD = 9.6$) for the matched changes, 122.2 files ($SD = 10.3$) for the high probability changes, 37.2 files ($SD = 10.4$) for low probability changes, and 11.7 ($SD = 8.2$) files for other code changes. See Figure 6.2.

We also found that changes categorized as matched or high probability changes have more related changes (on average 2.0 related changes for the matched changes and 1.8 related changes for the high probability changes) than the low probability or other changes (on average, 0.6 related changes for low probability changes, and 0.6 related changes for other code changes). In summary, our analysis indicates that changes that we relate to experiments are generally larger in terms of the files changed and have more related changes.

Bing can now use these results to identify the challenges that hindered developers from launching experiments. The challenges identified can then be addressed within the experimental framework.

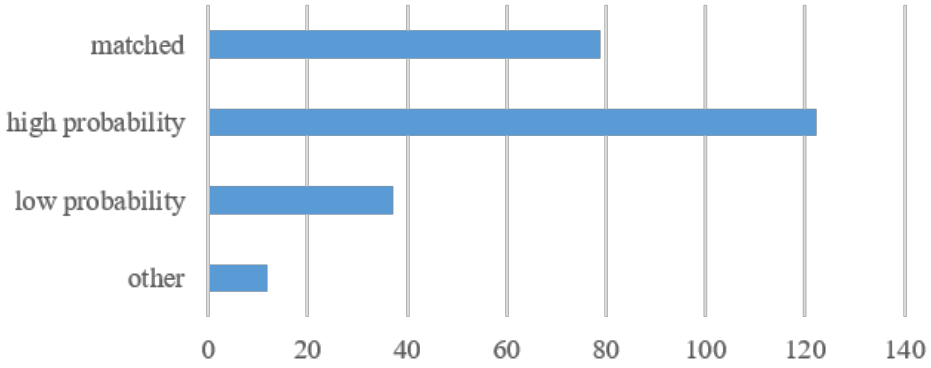


Figure 6.2: Average number of files for matched, high probability, low probability, and other code changes.

6.6 Success Rate of Experiments (RQ2)

> **RQ2**: What percentage of experiments are ultimately deployed to all users?

Our empirical analysis indicates that 33.4% of the experiment groups were ultimately deployed to all users. Our observation supports Kohavi et al. [Kohavi et al., 2009a] who reported that about a third of the experiments improve the

Table 6.1: Characteristics of experiments for the major Bing components included in our analysis. The exposure duration (*exp. dur.*) is calculated over all iterations (*iter.*) of an experiment, (contributors = *contr*).

Bing component	development efforts			experimentation time		
	# contr.	# files	# changes	exp. dur.	# iter.	# owners
Ads	2.2	12.1	4.2	29.1	1.9	6.2
Cortana	1.8	11.7	4.2	17.3	1.6	4.7
Datamining	1.6	9.4	3.9	19.7	1.9	3.1
Engagement	1.7	13.2	4.4	30.0	2.2	4.9
Index	2.0	15.5	4.2	11.5	1.6	3.8
Infrastructure	1.2	5.6	2.2	21.8	3.1	5.0
Local	2.2	14.0	5.2	25.4	1.7	4.2
Multimedia	1.9	18.3	4.1	15.0	1.5	5.3
Relevance	2.6	13.8	7.2	16.3	1.7	4.7
Segments	2.3	9.2	5.3	23.4	1.5	3.9
UX	2.1	15.3	4.9	22.9	2.2	4.9
Windows Search	1.7	16.2	3.0	14.6	1.5	4.8

metrics they were designed to improve. For 18% of the experiment groups, our procedure cannot infer whether the experiment was deployed to all users, these would require additional analysis to identify their status (see Section 6.4.2 for details). We also found considerable differences between the components within Bing. While components related to multimedia deploy 50.7% of the experiments to all customers, the rate is lower for components related to the index server (24.9%) for example. The varying rates of deployed experiments among components in Bing indicate that different components have different levels of difficulty to innovate enhancements which significantly improves the user experience. Bing developers mentioned that they are happy that they do not have a specific target of successful experiments, enabling them to try out

new ideas. We further observed that the percentage of non-deployed experiments is increasing over time. One possible root cause might be that it becomes more difficult to find a niche for innovation as the product matures. On the other hand, since ExP facilitates systematic experiments, developers might test different variants of the same feature in separately captured experiments. Through our analysis, Bing is enabled to analyze the metrics that lead to a data-driven decision. Knowing which metrics are crucial for a particular component, opens the possibility to further automate a data-driven decision and offer an improved scorecard interface.

6.7 Differences between Deployed and Non-Deployed Experiments (RQ3)

> **RQ3**: How do the experiments which are deployed to all users differ from the experiments which were not deployed to all users in terms of time spans, number of people involved, files and changes?

To answer RQ3, we opposed several characteristics that we captured for the deployed and non-deployed experiments. We did not observe significant differences for the experiments' exposure durations, number of iterations conducted within an experiment group, number of experiment owners, and number of metrics collected. We found differences in the way the code is developed for an experiment. A Welch two sample t-test indicates that experiments for which the treatment was ultimately deployed to all users have significantly more changes ($M = 5.1$) associated than treatments of experiments which have not been deployed ($M = 2.9$) at the time of the analysis, $t = -15.86, p < .001$. Furthermore, significantly more people contributed these changes for the deployed experiments ($M = 2.0$) than for the non-deployed experiments ($M = 1.6$), $t = -9.05, p < .001$. We also found that the code changes for deployed experiments are overall larger, in terms of the files that were changed ($M = 22.1$ for deployed experiments, $M = 14.2$ for non-deployed experiments, $t = -11.23, p < .001$), the unique files that were changed ($M = 14.4$ for deployed experiments, $M = 10.7$ for

non-deployed experiments, $t = -7.30, p < .001$), and the number of lines that were changed ($M = 690$ for deployed experiments, $M = 231$ for non-deployed experiments, $t = -2.58, p = .01$).

We can infer for experiments which were ultimately deployed to all users that the captured metrics allowed a data-driven decision. At this point of our analysis, we cannot infer for the experiments that were not deployed to all users whether the metrics indicated that the user experience is decreased or whether there was no significant difference observed between the treatment and the control. Nevertheless, our results indicate that the collaboration of more contributors leads to the fruitful execution of an experiment. To better understand whether the collaboration of more people causes more files being changed or whether the need to change more files requires more people to collaborate, is planned for future work. Understanding the differences between deployed and non-deployed experiments, teams may be able to identify which category of experiments are more likely to be more successful and which category of experiments may require more monitoring.

6.8 Threats to Validity

The *external validity* of our empirical analysis is threatened by the analysis of only one project. Because Bing is a mature large service, our results are not generalizable to less mature products. Furthermore, we also believe that the experimental process is more complex for on-premises products. However, the project which we analyzed comprises several inherently different components. We tried to mitigate this difference by considering each component separately. Furthermore, due to data consistency reasons we limited our analysis to experiments of the past 2.5 years.

The *internal validity* of our analysis is threatened by the fact that components within Bing use slightly different ways to capture, configure and deploy experiments. Bing is a composition of very large services that use different programming languages. Further, Bing is developed by hundreds of developers, who implement source code for experiments in different ways. Therefore, we

were limited in the development of parsers to link code changes to experiments and to infer whether experiments have been shipped. Hence, our analysis does not cover all experiments run within Bing, but presents an analysis on a subset of Bing's experiments. Furthermore, our analysis on the experiments' life-cycle does not capture the time spent on designing the experiment and analyzing the gathered user data. Our analysis is therefore a first approximation of the actual time needed to conduct controlled experiments in a large-scale software product.

6.9 Discussion

The opportunity to experiment with products drastically changed the way software is deployed within Bing. Our empirical analysis showed that experimentation has become an integral part within the deployment cycle. In the following, we discuss different aspects of the experimentation process and our planned future work.

6.9.1 Should Experimentation be Done for All Code Changes?

Bing has significantly increased the number of experiments since 2009 [Kohavi et al., 2013]. Further, many people are involved in the execution of an experiment who spend time preparing and executing experiments. The experimentation process is now a substantial part within the deployment cycle. We also observed that experiments can become a limiting factor of the cycle time within the deployment cycle, and hence we suggest that practice as well as research should not only focus on methods to accelerate the deployment of code changes, but on methods to identify experiments which are worthwhile to run and on methods to ensure that the experiment is run without issues.

We observed that generally larger code changes are linked to experiments. While it is possible to run a controlled experiment with each kind of change, we conclude that smaller changes have other priorities than improving the user experience. As an example, for a bug fix, the most important issues are to rapidly understand whether the deployed fix does not introduce further issues

and whether the change fixes the bug. For small code changes, users may be less likely to significantly react. On the other side of the spectrum, the difficulty in the experiment design, measurement, and analysis is substantially increased for large code changes as many different aspects about the larger change can influence user behaviors. Therefore, adapting experimentation means to understand the trade-off between running a controlled experiment and other means to verify a code change or to offer different experimentation processes for different kinds of changes. We believe that the cost of a controlled experiment could be dramatically decreased for bug fixes, if these changes could be deployed after a shorter amount of time and do not have to improve the user experience necessarily. Requiring a hypothesis for every change is too great an overhead for small changes, such as big fixes. Therefore, we believe that an experimentation process tailored to the different kinds of code changes may be more efficient.

6.9.2 Size of the Code Changes

We observed that code changes which we classified as matched changes or high probability changes have overall more development activity associated with them than changes which we classified as low probability or other changes. This observation raises the question whether experimentation in a mature system is only enabled by changes big enough to cause measurable effects on the end users of the product. On the other hand, developers may not want to spend additional time for experimentation if the change is reasonably small. Hence, we suggest that different experimentation processes and frameworks should be used for different kinds of changes. For future work, we plan to investigate further the relation between bigger releases and the amount of experiments run and compare these findings to experimentation activity in a less mature system. Furthermore, we plan to investigate how continuous experimentation influences the way developers work.

6.9.3 Developers as Data Analysts

We observed that on average 1409 metrics about the users' behavior for each version of the product are identified and analyzed by experimenters. Further, experimentation frameworks offer to analyze an ongoing experiment multiple times a day. The collected metrics are not always straight-forward to interpret, as Kohavi et al. [Kohavi et al., 2012] illustrate on five real-world examples. This difficulty of interpreting the collected metrics suggests an inevitable shift of traditional development work to rigorous data analyses. This observation agrees with the observations of Kim et al. [Kim et al., 2016] who found that data scientist are increasingly important within software development teams. As these data analyses have potential to impact the annual revenue of a product [Kohavi et al., 2012], proper data analyses is of superior importance. As Lindgren and Münch [Lindgren and Münch, 2015] found out when interviewing people of different roles in ten software companies, a lack of time and missing expertise were named as reasons of inadequate data analysis. This lack of data analysis expertise was also identified for experiments run in a B2B environment [Rissanen and Münch, 2015]. Hence, the team around the Experimentation Platform introduced one-day classes in statistics and experiment design, which nowadays even have wait lists [Kohavi et al., 2009a]. We plan to further investigate how developers can be supported in coping with the captured metrics about the user behavior, for example through improved user interfaces and summarized views.

6.9.4 Success of Experiments

The success of an experiment can be considered from different standpoints. In our analysis, we first verified whether the code change is eventually shipped to all users. We observed that experiments which were shipped are significantly larger and that more people contributed source code for the experiment. However, an experiment can also be considered successful if a data-driven decision of whether to ship or abandon a code change was enabled and hence a development team had the possibility to learn more about their users. In a next step, we plan to

analyze this learning value of experiments and we plan to figure out what the characteristics of experiments are that enable a data-driven decision.

6.10 Conclusion

The opportunity to experiment with a software product denotes a radical change in how software is deployed. While previously every change was deployed to all users, now only the changes which have a measurable improvement on the user experience are deployed to all users. We characterized such an experimentation process employed in a large-scale and mature product, i.e. Bing. We further analyzed 21,220 experiments over the past 2.5 years and observed that 33.4% of these experiments have been deployed to all users of the product. Our characterization of the experiments and their development activities revealed that experiments which are eventually shipped to all users, have generally more development activity.

Acknowledgment

We would like to thank the Experimentation Platform team for sharing historical data of experiments and helpful discussions of this work. We also thank the people who reviewed this paper, in particular the NCSU Realsearch research group, for providing valuable feedback.

Bibliography

[Adams and McIntosh, 2016] Adams, B. and McIntosh, S. (2016). Modern Release Engineering in a Nutshell – Why Researchers Should Care. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering*, SANER '16, pages 78–90.

[Altmann, 2001] Altmann, E. M. (2001). Near-term memory in programming: a simulation-based analysis. *International Journal of Human Computer Studies*, 54(2):189–210.

[Amann et al., 2016] Amann, S., Proksch, S., Nadi, S., and Mezini, M. (2016). A Study of Visual Studio Usage in Practice. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering*, SANER '16, pages 124–134.

[AmazonWebServices, 2017] AmazonWebServices (2017). Amazon EC2. `http://aws.amazon.com/ec2`. Accessed online: 2017-05-21.

[Augustine et al., 2015] Augustine, V., Francis, P., Qu, X., Shepherd, D., Snipes, W., Bräunlich, C., and Fritz, T. (2015). A Field Study on Fostering Structural Navigation with Prodet. In *Proceedings of the 37th International Conference on Software Engineering*, ICSE '15, pages 229–238.

[Bailey and Konstan, 2006] Bailey, B. P. and Konstan, J. A. (2006). On the Need for Attention-Aware Systems: Measuring Effects of Interruption on Task Performance, Error Rate, and Affective State. *Computers in Human Behavior*, 22(4):685 – 708. Special issue: Attention Aware Systems.

[Bakshy et al., 2014] Bakshy, E., Eckles, D., and Bernstein, M. S. (2014). Designing and Deploying Online Field Experiments. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14, pages 283–292.

[Barnett et al., 2015] Barnett, M., Bird, C., Brunet, J. a., and Lahiri, S. K. (2015). Helping Developers Help Themselves: Automatic Decomposition of Code Review Changesets. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 134–144.

[Basili et al., 1986] Basili, V. R., Selby, R. W., and Hutchens, D. H. (1986). Experimentation in Software Engineering. *IEEE Transactions on Software Engineering*, 12(7):733–743.

[Bednarik, 2012] Bednarik, R. (2012). Expertise-dependent Visual Attention Strategies Develop over Time During Debugging with Multiple Code Representations. *International Journal of Human-Computer Studies*, 70(2):143–155.

[Bednarik and Tukiainen, 2006] Bednarik, R. and Tukiainen, M. (2006). An Eye-tracking Methodology for Characterizing Program Comprehension Processes. In *Proceedings of the Symposium on Eye Tracking Research and Applications*, pages 125–132.

[Beller et al., 2015] Beller, M., Gousios, G., Panichella, A., and Zaidman, A. (2015). When, How, and Why Developers (Do Not) Test in Their IDEs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 179–190.

[Bettenburg et al., 2008] Bettenburg, N., Just, S., Schröter, A., Weiss, C., Premraj, R., and Zimmermann, T. (2008). What Makes a Good Bug Report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE '08, pages 308–318.

[Biegel et al., 2015] Biegel, B., Baltes, S., Scarpellini, I., and Diehl, S. (2015). CodeBasket: Making Developers' Mental Model Visible and Explorable. In *Proceedings of the 2nd International Workshop on Context for Software Development*, CSD '15, pages 20–24.

[Biggerstaff et al., 1994] Biggerstaff, T. J., Mitbander, B. G., and Webster, D. E. (1994). Program Understanding and the Concept Assignment Problem. *Communications of the ACM*, 37(5):72–82.

[Binkley et al., 2013] Binkley, D., Davis, M., Lawrie, D., Maletic, J. I., Morrell, C., and Sharif, B. (2013). The Impact of Identifier Style on Effort and Comprehension. *Empirical Software Engineering*, 18(2):219–276.

[Bosch, 2012] Bosch, J. (2012). Building products as innovation experiment systems. In Cusumano, M. A., Iyer, B., and Venkatraman, N., editors, *3rd International Conference on Software Business*, volume 114 of *Lecture Notes in Business Information Processing*, pages 27–39. Springer.

[Bragdon et al., 2010] Bragdon, A., Zeleznik, R., Reiss, S. P., Karumuri, S., Cheung, W., Kaplan, J., Coleman, C., Adeputra, F., and LaViola, Jr., J. J. (2010). Code Bubbles: A Working Set-based Interface for Code Understanding and Maintenance. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 2503–2512.

[Brooks, 1987] Brooks, Jr., F. P. (1987). No Silver Bullet Essence and Accidents of Software Engineering. *Computer*, 20(4):10–19.

[Brooks, 1978] Brooks, R. (1978). Using a Behavioral Theory of Program Comprehension in Software Engineering. In *Proceedings of the 3rd International Conference on Software Engineering*, ICSE '78, pages 196–201.

[Brooks, 1983] Brooks, R. (1983). Towards a Theory of the Comprehension of Computer Programs. *International Journal of Man-Machine Studies*, 18:543–554.

[Brooks, 1999] Brooks, R. (1999). Towards a Theory of the Cognitive Processes in Computer Programming. *International Journal of Human-Computer Studies*, 51(2):197 – 211.

[Buckner et al., 2005] Buckner, J., Buchta, J., Petrenko, M., and Rajlich, V. (2005). JRipples: A Tool for Program Comprehension During Incremental

Change. In *Proceedings of the 13th International Workshop on Program Comprehension*, IWPC '05, pages 149–152.

[Bugzilla, 2017] Bugzilla (2017). `https://bugs.eclipse.org/bugs/`. Accessed online: 2017-05-21.

[Coblenz et al., 2006] Coblenz, M. J., Ko, A. J., and Myers, B. A. (2006). JASPER: An Eclipse Plug-in to Facilitate Software Maintenance Tasks. In *OOPSLA Workshop on Eclipse Technology eXchange*, eclipse '06, pages 65–69.

[Coman and Sillitti, 2008] Coman, I. D. and Sillitti, A. (2008). Automated Identification of Tasks in Development Sessions. In *16th IEEE International Conference on Program Comprehension*, ICPC '08, pages 212–217.

[Corritore and Wiedenbeck, 1999] Corritore, C. L. and Wiedenbeck, S. (1999). Mental Representations of Expert Procedural and Object-Oriented Programmers in a Software Maintenance Task. *International Journal of Human-Computer Studies*, 50(1):61 – 83.

[Crook et al., 2009] Crook, T., Frasca, B., Kohavi, R., and Longbotham, R. (2009). Seven Pitfalls to Avoid when Running Controlled Experiments on the Web. In *Proceedings of the 15th ACM International Conference on Knowledge Discovery and Data Mining*, KDD '09, pages 1105–1114.

[Crosby and Stelovsky, 1990] Crosby, M. E. and Stelovsky, J. (1990). How Do We Read Algorithms? A Case Study. *Computer*, 23(1):24–35.

[Davenport, 2009] Davenport, T. H. (2009). How to Design Smart Business Experiments. *Harvard Business Review*, pages 69–76.

[De Smet et al., 2014] De Smet, B., Lempereur, L., Sharafi, Z., Guéhéneuc, Y.-G., Antoniol, G., and Habra, N. (2014). Taupe: Visualizing and Analyzing Eye-tracking Data. *Science of Computer Programming*, 79:260–278.

[DeLine et al., 2005a] DeLine, R., Czerwinski, M., and Robertson, G. (2005a). Easing program comprehension by sharing navigation data. In *Proceedings of*

the Symposium on Visual Languages and Human-Centric Computing, VL/HCC '05, pages 241–248.

[DeLine et al., 2005b] DeLine, R., Khella, A., Czerwinski, M., and Robertson, G. (2005b). Towards Understanding Programs Through Wear-based Filtering. In *Proceedings of the Symposium on Software Visualization*, SoftVis '05, pages 183–192.

[Deng, 2015] Deng, A. (2015). Objective Bayesian Two Sample Hypothesis Testing for Online Controlled Experiments. In *Proceedings of the 24th International Conference on World Wide Web (Companion)*, WWW '15 Companion, pages 923–928.

[Deng et al., 2013] Deng, A., Xu, Y., Kohavi, R., and Walker, T. (2013). Improving the Sensitivity of Online Controlled Experiments by Utilizing Pre-experiment Data. In *Proceedings of the 6th ACM International Conference on Web Search and Data Mining*, WSDM '13, pages 123–132.

[Duvall et al., 2007] Duvall, P., Matyas, S., and Glover, A. (2007). *Continuous Integration: Improving Software Quality and Reducing Risk*. A Martin Fowler signature book. Addison-Wesley.

[EclipseSWT, 2017] EclipseSWT (2017). The Standard Widget Toolkit. `http://eclipse.org/swt`. Accessed online: 2017-05-21.

[Fagerholm et al., 2014] Fagerholm, F., Guinea, A. S., Mäenpää, H., and Münch, J. (2014). Building Blocks for Continuous Experimentation. In *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*, RCoSE '14, pages 26–35.

[Fagerholm et al., 2017] Fagerholm, F., Guinea, A. S., Mäenpää, H., and Münch, J. (2017). The RIGHT model for Continuous Experimentation. *Journal of Systems and Software*, 123:292 – 305.

[Field, 2005] Field, A. (2005). *Discovering Statistics Using SPSS*. SAGE Publications.

[Fluri et al., 2007] Fluri, B., Wuersch, M., PInzger, M., and Gall, H. (2007). Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743.

[FreeMind, 2017] FreeMind (2017). A Premier Mind-Mapping Software Written in Java. `http://sourceforge.net/projects/freemind/`. Accessed online: 2017-05-21.

[FreeMindBug, 2017] FreeMindBug (2017). `https://sourceforge.net/p/freemind/bugs/1063/`. Accessed online: 2017-05-21.

[Fritz et al., 2014a] Fritz, T., Begel, A., Müller, S. C., Yigit-Elliott, S., and Züger, M. (2014a). Using Psycho-physiological Measures to Assess Task Difficulty in Software Development. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE '14, pages 402–413.

[Fritz et al., 2014b] Fritz, T., Shepherd, D. C., Kevic, K., Snipes, W., and Bräunlich, C. (2014b). Developers' Code Context Models for Change Tasks. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '14, pages 7–18.

[GoogleAnalytics, 2016] GoogleAnalytics (2016). Experiments - articles & solutions. `https://developers.google.com/analytics/solutions/experiments`. Accessed online: 2016-10-21.

[Gson, 2016] Gson (2016). A Java Serialization/Deserialization Library to Convert Java Objects into JSON and Back. `https://github.com/google/gson`. Accessed online: 2017-01-06.

[Haiduc et al., 2010a] Haiduc, S., Aponte, J., and Marcus, A. (2010a). Supporting Program Comprehension with Source Code Summarization. In *Proceedings of the 32nd International Conference on Software Engineering*, ICSE '10, pages 223–226.

[Haiduc et al., 2010b] Haiduc, S., Aponte, J., Moreno, L., and Marcus, A. (2010b). On the Use of Automated Text Summarization Techniques for

Summarizing Source Code. In *Proceedings of the 17th Working Conference on Reverse Engineering*, WCRE '10, pages 35–44.

[Hill et al., 2007] Hill, E., Pollock, L., and Vijay-Shanker, K. (2007). Exploring the Neighborhood with Dora to Expedite Software Maintenance. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 14–23.

[Humble and Farley, 2010] Humble, J. and Farley, D. (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Signature Series (Fowler). Pearson Education.

[Iqbal et al., 2004] Iqbal, S. T., Adamczyk, P. D., Zheng, X. S., and Bailey, B. P. (2004). Changes in Mental Workload During Task Execution. In *Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology*, UIST '04.

[iTrace, 2015] iTrace (2015). `www.csis.ysu.edu/~bsharif/itraceMylyn`. Accessed online: 2015-03-15.

[JabRef, 2015] JabRef (2015). `http://www.jabref.sourceforge.net`. Accessed online: 2015-03-15.

[Janzen and De Volder, 2003] Janzen, D. and De Volder, K. (2003). Navigating and Querying Code Without Getting Lost. In *Proceedings of the 2nd International Conference on Aspect-oriented Software Development*, AOSD '03, pages 178–187.

[JavaPasswordSafe, 2017] JavaPasswordSafe (2017). Password Safe Java Clone in Native Look & Feel. `http://sourceforge.net/projects/jpwsafe/`. Accessed online: 2017-05-21.

[JavaPasswordSafeBug, 2017] JavaPasswordSafeBug (2017). `https://sourceforge.net/p/jpwsafe/bugs/30/`. Accessed online: 2017-05-21.

[Just and Carpenter, 1980] Just, M. and Carpenter, P. (1980). A Theory of Reading: From Eye Fixations to Comprehension. *Psychological Review*, 87:329–354.

[Kersten and Murphy, 2005] Kersten, M. and Murphy, G. C. (2005). Mylar: A Degree-of-interest Model for IDEs. In *Proceedings of the 4th International Conference on Aspect-oriented Software Development*, AOSD '05, pages 159–168.

[Kersten and Murphy, 2006] Kersten, M. and Murphy, G. C. (2006). Using Task Context to Improve Programmer Productivity. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE '06, pages 1–11.

[Kevic and Fritz, 2014] Kevic, K. and Fritz, T. (2014). A Dictionary to Translate Change Tasks to Source Code. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR '14, pages 320–323.

[Kevic et al., 2014] Kevic, K., Fritz, T., and Shepherd, D. C. (2014). CoMoGen: An Approach to Locate Relevant Task Context by Combining Search and Navigation. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 61–70.

[Kevic et al., 2017] Kevic, K., Walters, B., Shaffer, T., Sharif, B., Shepherd, D., and Fritz, T. (2017). Eye Gaze and Interaction Contexts for Change Tasks – Observations and Potential. *Journal of Systems and Software*, 128:252 – 266.

[Kevic et al., 2015] Kevic, K., Walters, B. M., Shaffer, T. R., Sharif, B., Shepherd, D. C., and Fritz, T. (2015). Tracing Software Developers' Eyes and Interactions for Change Tasks. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE '15, pages 202–213.

[Kiczales et al., 1997] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). *Aspect-oriented Programming*, pages 220–242. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Kim et al., 2016] Kim, M., Zimmermann, T., DeLine, R., and Begel, A. (2016). The Emerging Role of Data Scientists on Software Development Teams. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 96–107.

[Ko et al., 2005] Ko, A. J., Aung, H., and Myers, B. A. (2005). Eliciting Design Requirements for Maintenance-oriented IDEs: A Detailed Study of Corrective and Perfective Maintenance Tasks. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 126–135.

[Ko et al., 2007] Ko, A. J., DeLine, R., and Venolia, G. (2007). Information Needs in Collocated Software Development Teams. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 344–353.

[Ko and Myers, 2005] Ko, A. J. and Myers, B. A. (2005). A Framework and Methodology for Studying the Causes of Software Errors in Programming Systems. *Journal of Visual Languages and Computing*, 16(1-2):41–84.

[Ko et al., 2006] Ko, A. J., Myers, B. A., Coblenz, M. J., and Aung, H. H. (2006). An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information During Software Maintenance Tasks. *IEEE Transactions on Software Engineering*, 32(12):971–987.

[Kohavi et al., 2009a] Kohavi, R., Crook, T., Longbotham, R., Frasca, B., Henne, R., Ferres, J. L., and Melamed, T. (2009a). Online Experimentation at Microsoft. In *Proceedings of the 3rd Workshop on Data Mining Case Studies and Practice Prize*, DMCS III, pages 11–22.

[Kohavi et al., 2012] Kohavi, R., Deng, A., Frasca, B., Longbotham, R., Walker, T., and Xu, Y. (2012). Trustworthy Online Controlled Experiments: Five Puzzling Outcomes Explained. In *Proceedings of the 18th ACM International Conference on Knowledge Discovery and Data Mining*, KDD '12, pages 786–794.

[Kohavi et al., 2013] Kohavi, R., Deng, A., Frasca, B., Walker, T., Xu, Y., and Pohlmann, N. (2013). Online Controlled Experiments at Large Scale. In

Proceedings of the 19th ACM International Conference on Knowledge Discovery and Data Mining, KDD '13, pages 1168–1176.

[Kohavi et al., 2014] Kohavi, R., Deng, A., Longbotham, R., and Xu, Y. (2014). Seven Rules of Thumb for Web Site Experimenters. In *Proceedings of the 20th ACM International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 1857–1866.

[Kohavi and Longbotham, 2016] Kohavi, R. and Longbotham, R. (2016). Online Controlled Experiments and A/B Testing. In Sammut, C. and Webb, G. I., editors, *Encyclopedia of Machine Learning and Data Mining*, pages 1–8, Boston, MA. Springer US.

[Kohavi et al., 2009b] Kohavi, R., Longbotham, R., Sommerfield, D., and Henne, R. M. (2009b). Controlled Experiments on the Web: Survey and Practical Guide. *Data Mining and Knowledge Discovery*, 18(1):140–181.

[Korel and Laski, 1988] Korel, B. and Laski, J. (1988). Dynamic Program Slicing. *Information Processing Letters*, 29(3):155–163.

[LaToza et al., 2007] LaToza, T. D., Garlan, D., Herbsleb, J. D., and Myers, B. A. (2007). Program Comprehension As Fact Finding. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '07, pages 361–370.

[LaToza and Myers, 2011] LaToza, T. D. and Myers, B. A. (2011). Visualizing Call Graphs. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, VL/HCC '11, pages 117–124.

[LaToza et al., 2014] LaToza, T. D., Towne, W. B., Adriano, C. M., and van der Hoek, A. (2014). Microtask Programming: Building Software with a Crowd. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, UIST '14, pages 43–54.

[LaToza et al., 2006] LaToza, T. D., Venolia, G., and DeLine, R. (2006). Maintaining Mental Models: A Study of Developer Work Habits. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 492–501.

[Lawrance et al., 2007] Lawrance, J., Bellamy, R., and Burnett, M. (2007). Scents in Programs: Does Information Foraging Theory Apply to Program Maintenance? In *IEEE Symposium on Visual Languages and Human-Centric Computing*, VL/HCC '07, pages 15–22.

[Lawrance et al., 2008] Lawrance, J., Bellamy, R., Burnett, M., and Rector, K. (2008). Using Information Scent to Model the Dynamic Foraging Behavior of Programmers in Maintenance Tasks. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, pages 1323–1332.

[Lee et al., 2011] Lee, T., Nam, J., Han, D., Kim, S., and In, H. P. (2011). Micro Interaction Metrics for Defect Prediction. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 311–321.

[Lindgren and Münch, 2015] Lindgren, E. and Münch, J. (2015). *Software Development as an Experiment System: A Qualitative Survey on the State of the Practice*, pages 117–128. Springer International Publishing, Cham.

[Mason et al., 1989] Mason, R., Gunst, R., and Hess, J. (1989). *Statistical Design and Analysis of Experiments: with Applications to Engineering and Science*. Wiley series in probability and mathematical statistics: Applied probability and statistics. Wiley.

[McMillan et al., 2011] McMillan, C., Grechanik, M., Poshyvanyk, D., Xie, Q., and Fu, C. (2011). Portfolio: Finding Relevant Functions and Their Usage. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 111–120.

[Mindell, 2008] Mindell, D. (2008). *Digital Apollo: Human and Machine in Spaceflight*. Inside Technology Series. MIT Press.

[Minelli et al., 2015] Minelli, R., and, A. M., and Lanza, M. (2015). I Know What You Did Last Summer: An Investigation of How Developers Spend Their Time. In *Proceedings of the 23rd International Conference on Program Comprehension*, ICPC '15, pages 25–35.

[Minelli et al., 2016] Minelli, R., Mocci, A., Robbes, R., and Lanza, M. (2016). Taming the IDE with Fine-Grained Interaction Data. In *2016 IEEE 24th International Conference on Program Comprehension*, ICPC '16, pages 1–10.

[Moreno et al., 2013] Moreno, L., Aponte, J., Sridhara, G., Marcus, A., Pollock, L., and Vijay-Shanker, K. (2013). Automatic Generation of Natural Language Summaries for Java Classes. In *Proceedings of the 21st International Conference on Program Comprehension*, ICPC '13, pages 23–32.

[Müller and Fritz, 2015] Müller, S. C. and Fritz, T. (2015). Stuck and Frustrated or in Flow and Happy: Sensing Developers' Emotions and Progress. In *Proceedings of the 37th International Conference on Software Engineering*, ICSE '15, pages 688–699.

[Müller and Fritz, 2016] Müller, S. C. and Fritz, T. (2016). Using (Bio)Metrics to Predict Code Quality Online. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 452–463.

[Murphy et al., 2006] Murphy, G. C., Kersten, M., and Findlater, L. (2006). How Are Java Software Developers Using the Eclipse IDE? *IEEE Software*, 23(4):76–83.

[Murphy et al., 2005] Murphy, G. C., Kersten, M., Robillard, M. P., and Čubranić, D. (2005). The Emergent Structure of Development Tasks. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, ECOOP'05, pages 33–48.

[Mylyn, 2015] Mylyn (2015). `eclipse.org/mylyn/`. Accessed online: 2015-03-15.

[Nan and Harter, 2009] Nan, N. and Harter, D. E. (2009). Impact of Budget and Schedule Pressure on Software Development Cycle Time and Effort. *IEEE Transactions on Software Engineering*, 35(5):624–637.

[Negara et al., 2013] Negara, S., Chen, N., Vakilian, M., Johnson, R. E., and Dig, D. (2013). A Comparative Study of Manual and Automated Refactorings. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 552–576, Berlin, Heidelberg. Springer-Verlag.

[Negara et al., 2012] Negara, S., Vakilian, M., Chen, N., Johnson, R. E., and Dig, D. (2012). Is It Dangerous to Use Version Control Histories to Study Source Code Evolution? In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 79–103, Berlin, Heidelberg. Springer-Verlag.

[Olsson et al., 2012] Olsson, H. H., Alahyari, H., and Bosch, J. (2012). Climbing the "Stairway to Heaven" – A Mulitiple-Case Study Exploring Barriers in the Transition from Agile Development Towards Continuous Deployment of Software. In *Proceedings of the 38th Euromicro Conference on Software Engineering and Advanced Applications*, SEAA '12, pages 392–399.

[Panichella et al., 2016] Panichella, S., Panichella, A., Beller, M., Zaidman, A., and Gall, H. C. (2016). The Impact of Test Case Summaries on Bug Fixing Performance: An Empirical Investigation. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 547–558.

[Parnin and Gorg, 2006] Parnin, C. and Gorg, C. (2006). Building Usage Contexts During Program Comprehension. In *Proceedings of the 14th IEEE International Conference on Program Comprehension*, ICPC '06, pages 13–22.

[Parnin et al., 2017] Parnin, C., Helms, E., Atlee, C., Boughton, H., Ghattas, M., Glover, A., Holman, J., Micco, J., Murphy, B., Savor, T., Stumm, M., Whitaker, S., and Williams, L. (2017). The Top 10 Adages in Continuous Deployment. *IEEE Software*, 34(3):86–95.

[Pennington, 1987] Pennington, N. (1987). Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. *Cognitive Psychology*, 19(3):295 – 341.

[Perry et al., 1994] Perry, D. E., Staudenmayer, N., and Votta, L. G. (1994). People, Organizations, and Process Improvement. *IEEE Software*, 11(4):36–45.

[Piorkowski et al., 2012] Piorkowski, D., Fleming, S., Scaffidi, C., Bogart, C., Burnett, M., John, B., Bellamy, R., and Swart, C. (2012). Reactive Information Foraging: An Empirical Investigation of Theory-based Recommender Systems for Programmers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '12, pages 1471–1480.

[Piorkowski et al., 2011] Piorkowski, D., Fleming, S. D., Scaffidi, C., John, L., Bogart, C., John, B. E., Burnett, M., and Bellamy, R. (2011). Modeling Programmer Navigation: A Head-to-Head Empirical Evaluation of Predictive Models. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, VL/HCC '11, pages 109–116.

[PlanOut, 2016] PlanOut (2016). A framework for online field experiments. `https://facebook.github.io/planout/`. Accessed: 2016-10-21.

[Ponzanelli et al., 2014] Ponzanelli, L., Bavota, G., Penta, M. D., Oliveto, R., and Lanza, M. (2014). Mining StackOverflow to Turn the IDE into a Self-confident Programming Prompter. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR '14, pages 102–111.

[Rachota, 2017] Rachota (2017). Rachota is a Portable Application for Time-tracking Different Projects. `http://sourceforge.net/projects/rachota/`. Accessed online: 2017-05-21.

[RachotaBug, 2017] RachotaBug (2017). `https://sourceforge.net/p/rachota/bugs/116/`. Accessed: 2017-04-29.

[Rayner, 1998] Rayner, K. (1998). Eye Movements in Reading and Information Processing: 20 Years of Research. *Psychological Bulletin*, 124(3):372.

[Ries, 2011] Ries, E. (2011). *The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*. Crown Publishing Group.

[Rissanen and Münch, 2015] Rissanen, O. and Münch, J. (2015). Continuous Experimentation in the B2B Domain: A Case Study. In *Proceedings of the 2nd International Workshop on Rapid Continuous Software Engineering*, RCoSE '15, pages 12–18.

[Rist, 1986] Rist, R. S. (1986). Plans in Programming: Definition, Demonstration, and Development. In *Papers Presented at the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers*, pages 28–47.

[Robbes and Lanza, 2008] Robbes, R. and Lanza, M. (2008). SpyWare: A Change-aware Development Toolset. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 847–850.

[Robillard, 2005] Robillard, M. P. (2005). Automatic Generation of Suggestions for Program Investigation. *ACM SIGSOFT Software Engineering Notes*, 30(5):11–20.

[Robillard et al., 2004] Robillard, M. P., Coelho, W., and Murphy, G. C. (2004). How Effective Developers Investigate Source Code: An Exploratory Study. *IEEE Transactions on Software Engineering*, 30(12):889–903.

[Robillard and Murphy, 2003] Robillard, M. P. and Murphy, G. C. (2003). Automatically inferring concern code from program investigation activities. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, ASE '03, pages 225–234.

[Robillard and Weigand-Warr, 2005] Robillard, M. P. and Weigand-Warr, F. (2005). ConcernMapper: Simple View-based Separation of Scattered Concerns. In *Proceedings of the OOPSLA Workshop on Eclipse Technology eXchange*, eclipse '05, pages 65–69.

[Robillard and Murphy, 2002] Robillard, M. R. and Murphy, G. C. (2002). Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 406–416.

[Rodeghero et al., 2014] Rodeghero, P., McMillan, C., McBurney, P. W., Bosch, N., and D'Mello, S. (2014). Improving Automated Source Code Summarization via an Eye-tracking Study of Programmers. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE '14, pages 390–401.

[Safer and Murphy, 2007] Safer, I. and Murphy, G. C. (2007). Comparing Episodic and Semantic Interfaces for Task Boundary Identification. In *Proceedings of the 2007 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '07, pages 229–243.

[Sando, 2017] Sando (2017). Instant Project Search Built on Lucene. `http://sando.codeplex.com/`. Accessed online: 2017-05-21.

[Seaman, 1999] Seaman, C. B. (1999). Qualitative Methods in Empirical Studies of Software Engineering. *IEEE Transactions on Software Engineering*, 25(4):557–572.

[Shaffer et al., 2015] Shaffer, T. R., Wise, J. L., Walters, B. M., Müller, S. C., Falcone, M., and Sharif, B. (2015). iTrace: Enabling Eye Tracking on Software Artifacts Within the IDE to Support Software Engineering Tasks. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE '15, pages 954–957.

[Sharif et al., 2013] Sharif, B., Jetty, G., Aponte, J., and Parra, E. (2013). An Empirical Study Assessing the Effect of Seeit 3D on Comprehension. In *Proceedings of the 1st Working Conference on Software Visualization*, VISSOFT '13, pages 1–10.

[Sharif and Kagdi, 2011] Sharif, B. and Kagdi, H. (2011). On the Use of Eye Tracking in Software Traceability. In *Proceedings of the 6th International Workshop on Traceability in Emerging Forms of Software Engineering*, TEFSE '11, pages 67–70.

[Sharif and Maletic, 2010a] Sharif, B. and Maletic, J. I. (2010a). An Eye Tracking Study on camelCase and under_score Identifier Styles. In *Proceedings*

of the 18th International Conference on Program Comprehension, ICPC '10, pages 196–205.

[Sharif and Maletic, 2010b] Sharif, B. and Maletic, J. I. (2010b). An Eye Tracking Study on the Effects of Layout in Understanding the Role of Design Patterns. In *Proceedings of the International Conference on Software Maintenance*, ICSM '10, pages 1–10.

[Shneiderman and Mayer, 1979] Shneiderman, B. and Mayer, R. (1979). Syntactic Semantic Interactions in Programmer Behavior: A Model and Experimental Results. *International Journal of Computer and Information Sciences*, 18:219–238.

[Siegmund et al., 2014] Siegmund, J., Kästner, C., Apel, S., Parnin, C., Bethmann, A., Leich, T., Saake, G., and Brechmann, A. (2014). Understanding Understanding Source Code with Functional Magnetic Resonance Imaging. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE '14, pages 378–389.

[Sillito et al., 2006] Sillito, J., Murphy, G. C., and De Volder, K. (2006). Questions Programmers Ask During Software Evolution Tasks. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '06, pages 23–34.

[Sillito et al., 2005] Sillito, J., Voider, K. D., Fisher, B., and Murphy, G. (2005). Managing Software Change Tasks: an Exploratory Study. In *Proceedings of the International Symposium on Empirical Software Engineering*, pages 23–32.

[Singer et al., 2005] Singer, J., Elves, R., and Storey, M. A. (2005). NavTracks: Supporting Navigation in Software Maintenance. In *Proceedings of the 21st International Conference on Software Maintenance*, ICSM '05, pages 325–334.

[Singer et al., 1997] Singer, J., Lethbridge, T., Vinson, N., and Anquetil, N. (1997). An Examination of Software Engineering Work Practices. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '97, pages 21–36.

[Snipes et al., 2014] Snipes, W., Nair, A. R., and Murphy-Hill, E. (2014). Experiences Gamifying Developer Adoption of Practices and Tools. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion '14, pages 105–114.

[Starke et al., 2009] Starke, J., Luce, C., and Sillito, J. (2009). Searching and skimming: An exploratory study. In *Proceedings of the International Conference on Software Maintenance*, ICSM '09, pages 157–166.

[Storey et al., 1999] Storey, M.-A. D., Fracchia, F. D., and Müller, H. A. (1999). Cognitive Design Elements to Support the Construction of a Mental Model During Software Exploration. *Journal of Systems and Software*, 44(3):171–185.

[Storey et al., 1997] Storey, M.-A. D., Wong, K., and Muller, H. A. (1997). How Do Program Understanding Tools Affect How Programmers Understand Programs. In *Proceedings of the 4th Working Conference on Reverse Engineering*, WCRE '97, pages 12–21.

[StudyArtifactsCoMoGen, 2017] StudyArtifactsCoMoGen (2017). Study Artifacts: Supporting Search and Navigation through Code Context Models. `https://github.com/abb-iss/Study-Artifacts-for-Code-Context-Models`. Accessed online: 2017-05-21.

[Stumpf et al., 2005] Stumpf, S., Bao, X., Dragunov, A., Dietterich, T. G., Herlocker, J., Li, L., and Shen, J. (2005). Predicting User Tasks: I Know What You're Doing. In *In 20th National Conference on Artificial Intelligence, Workshop on Human Comprehensible Machine Learning*, AAAI '05. Press.

[Tang et al., 2010] Tang, D., Agarwal, A., O'Brien, D., and Meyer, M. (2010). Overlapping Experiment Infrastructure: More, Better, Faster Experimentation. In *Proceedings of the 16th Conference on Knowledge Discovery and Data Mining*, KDD '10, pages 17–26.

[Tobii, 2015] Tobii (2015). `www.tobii.com/`. Accessed online: 2015-03-15.

[Turner et al., 2014] Turner, R., Falcone, M., Sharif, B., and Lazar, A. (2014). An Eye-tracking Study Assessing the Comprehension of C++ and Python Source Code. In *Proceedings of the Symposium on Eye Tracking Research and Applications*, ETRA '14, pages 231–234.

[Uwano et al., 2006] Uwano, H., Nakamura, M., Monden, A., and Matsumoto, K. (2006). Analyzing Individual Performance of Source Code Review Using Reviewers' Eye Movement. In *Proceedings of the Symposium on Eye Tracking Research and Applications*, ETRA '06, pages 133–140.

[Vans et al., 1999] Vans, A. M., von Mayrhauser, A., and Somlo, G. (1999). Program understanding behavior during corrective maintenance of large-scale software. *International Journal of Human-Computer Studies*, 51(1):31 – 70.

[Čubranić and Murphy, 2003] Čubranić, D. and Murphy, G. C. (2003). Hipikat: Recommending Pertinent Software Development Artifacts. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 408–418.

[von Mayrhauser and Vans, 1994] von Mayrhauser, A. and Vans, A. M. (1994). Comprehension Processes During Large Scale Maintenance. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, pages 39–48.

[Walters et al., 2013] Walters, B., Falcone, M., Shibble, A., and Sharif, B. (2013). Towards an Eye-Tracking Enabled IDE for Software Traceability Tasks. In *Proceedings of the 7th International Workshop on Traceability in Emerging Forms of Software Engineering*, TEFSE '13, pages 51–54.

[Walters et al., 2014] Walters, B., Shaffer, T., Sharif, B., and Kagdi, H. (2014). Capturing Software Traceability Links from Developers' Eye Gazes. In *Proceedings of the 22nd International Conference on Program Comprehension*, ICPC '14, pages 201–204.

[Wang et al., 2011] Wang, J., Peng, X., Xing, Z., and Zhao, W. (2011). An Exploratory Study of Feature Location Process: Distinct Phases, Recurring

Patterns, and Elementary Actions. In *Proceedings of the 27th International Conference on Software Maintenance*, ICSM '11, pages 213–222.

[Weiser, 1981] Weiser, M. (1981). Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449.

[Xu et al., 2015] Xu, Y., Chen, N., Fernandez, A., Sinno, O., and Bhasin, A. (2015). From Infrastructure to Culture: A/B Testing Challenges in Large Scale Social Networks. In *Proceedings of the 21th International Conference on Knowledge Discovery and Data Mining*, KDD '15, pages 2227–2236.

[Ying et al., 2004] Ying, A. T. T., Murphy, G. C., Ng, R., and Chu-Carroll, M. C. (2004). Predicting Source Code Changes by Mining Change History. *IEEE Transactions on Software Engineering*, 30(9):574–586.

[Yusuf et al., 2007] Yusuf, S., Kagdi, H., and Maletic, J. I. (2007). Assessing the Comprehension of UML Class Diagrams via Eye Tracking. In *Proceedings of the 15th International Conference on Program Comprehension*, ICPC '07, pages 113–122.

[Zimmermann et al., 2004] Zimmermann, T., Weisgerber, P., Diehl, S., and Zeller, A. (2004). Mining Version Histories to Guide Software Changes. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 563–572.

[Zimmermann et al., 2005] Zimmermann, T., Weissgerber, P., Diehl, S., and Zeller, A. (2005). Mining Version Histories to Guide Software Changes. *IEEE Transactions on Software Engineering*, 31(6):429–445.

[Zou and Godfrey, 2012] Zou, L. and Godfrey, M. W. (2012). An industrial case study of Coman's automated task detection algorithm: What Worked, What Didn't, and Why. In *Proceedings of the 28th International Conference on Software Maintenance*, ICSM '12, pages 6–14.

[Züger and Fritz, 2015] Züger, M. and Fritz, T. (2015). Interruptibility of Software Developers and Its Prediction Using Psycho-Physiological Sensors. In

Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '15, pages 2981–2990.

Curriculum Vitae

Personal Information

Name	Katja Kevic
Nationality	Switzerland
Date of Birth	August 3, 1986
Place of Birth	Schlieren, Switzerland

Education

2014 – 2017	PhD in Informatics Department of Informatics University of Zurich, Switzerland
2011 – 2013	Master of Science in Computer Science Department of Informatics University of Zurich, Switzerland
2007 – 2011	Bachelor of Science in Informatics Department of Informatics University of Zurich, Switzerland